SURVIVING AND THRIVING IN A MULTI-CORE WORLD

AN EBOOK FROM AMD DEVELOPER CENTRAL

November 2007

Trademarks

Table of Contents

Driving in the Fast Lane: Multi-Core Computing for Programmers, Part 1

Learn how applications and operating systems behave on multi-core systems.

by Alan Zeichick

The main road near my house, called Skyline Drive, drives me nuts. For several miles, it's a quasi-limited access highway. But for some inexplicable reason, it keeps alternating between one and two lanes in each direction. In the two-lane part, traffic moves along swiftly, even during rush hour. In the one-lane part, the traffic merges back together, and everything crawls to a standstill. When the next two-lane part appears, things speed up again.

Two lanes are better than one—and not just because they can accommodate twice as many cars. What makes the two-lane section better is that people can overtake. In the one-lane portion (which has a double-yellow line, so there's no passing), traffic is limited to the slowest truck's speed, or to little-old-man-peering-over-the-steering-wheel-of-his-Dodge-Dart speed. Wake me when we get there. But in the two-lane section, the traffic can sort itself out. Trucks move to the right, cars pass on the left. Police and other priority traffic weave in and out, using both lanes depending on which has more capacity at any particular moment. Delivery services with a convoy of trucks will exploit both lanes to improve throughput. The entire system becomes more efficient, and net flow of cars through those two-lane sections is considerably higher.

Okay, you've figured out that this is all about dual-core and multi-core computing, where cars are analogous to application threads, and the lanes are analogous to processor cores.

I'll have to admit that my analogy is somewhat simplistic, and purists will say that it's flawed, because an operating system has more flexibility to schedule tasks in a single-core environment under a preemptive multiprocessing environment. But that flexibility comes at a cost. Yes, if I were really modeling a microprocessor using Skyline Drive, cars would be able to pass each other in the single-lane section, but only if the car in front were to pull over and stop.

No, Cores Aren't Really Cars

Okay, enough about cars. Let's talk about dual-core and multi-core systems, why businesses are interested in buying them, and what implications all that should have for software developers like us.

A core is the part of a microprocessor that does all the computing. It comprises a central processing unit (the thing that executes the instruction set), as well as the L1 and L2 instruction and data caches. In a typical single-core Opteron processor's silicon, you have one core, as well as a memory controller and HyperTransport bus interfaces. The core is only able to handle one task at a time.

In a multi-core chip, you have multiple cores—the central processing unit and L1/L2 caches—on that piece of silicon. Each core can process one task at a time. However, you still only have one integrated memory controller and one set of HyperTransport bus interfaces. Connecting them all together is a crossbar switch, which gives each core equal access to the memory controllers and I/O subsystem.

The benefit of a multi-core system is that the microprocessor can handle multiple threads simultaneously, and those threads can be completely independent. In that sense, it's like northbound Skyline Drive, which can handle multiple lanes of traffic. Thanks to the crossbar switch and integrated memory controller, each core can access other system resources without disturbing what's going on in the other core (as if traffic in the left-hand lane can make a right turn without first moving into the right-hand lane).

It's best to consider a dual-core processor as an upgrade for a single-core processor, because you get twice as many threads being processed. A single-processor system with a dual-core chip can handle two hardware threads—just like a dual-processor single-core chip. Also, the efficient design of the internal crossbar switch in AMD's Opteron design means that if two threads need to talk to each other, they can do so inside the silicon, therefore data can be exchanged very rapidly. This works today in single-processor dual-core systems; within a few years, AMD says it will be able to do this within multi-processor systems as well. (By contrast, the Northbridge design used by Intel's dual-core Xeon processors means that the cores within the silicon can't talk to each other directly; they have to talk using the slower external front-side bus.)

You probably shouldn't think of a dual-core processor as a replacement for two single-core processors. In other words, it's not a good idea to replace a four-processor, single-core server with a dual-processor, dual-core server. While it might seem that you'd have the same performance—after all, you would have as many cores (four), that's not the case—there are two drawbacks.

First, in order to reduce power consumption and cooling, each of the cores running inside a dual-core processor is clocked slightly slower than an equivalent single-core Opteron processor. Thus, today you can have a piece of silicon that's a 2.6GHz single-core Opteron chip, or a 2.2GHz dual-core chip. It's likely that the dual-core chips will always be clocked slower than single-core; when quad-core chips come out, they'll be clocked slower still. (To use my car analogy, the dual-lane road has a lower speed limit than the single-lane road, in order to reduce net auto emissions on that stretch of highway.)

The other drawback exists because there's only a single shared memory controller, and a single set of shared HyperTransport interfaces—two cores running flat-out will have less memory and I/O bandwidth (to continue the car analogy: on a multi-lane highway, on-ramps and off-ramps are single-lane roads.) Most of the time, there's ample I/O bandwidth, so this isn't a bottleneck—but it's still something to consider.

The upshot of all this is that "conventional wisdom" says that the fastest dual-core processor won't actually run twice as fast as the fastest single core chip; in the real world, it'll produce a speed improvement of about 1.5 to 1.7 times the performance, depending on the application's I/O requirements. Again, that's why you should think of a dual-core chip as an upgrade for a single-core chip, not as a replacement for two single-core chips.

What This Means for Developers

In our car world, imagine a lot of the traffic on Skyline Drive was municipal: police and fire trucks, trash trucks, and street sweepers. That's the case on a typical PC or server: The operating system produces a huge number of threads. A modern operating system, like Linux, 32-bit or 64-bit Windows, or Solaris is inherently multithreaded: A process, such as the network listener, print spooler, or keyboard monitor, is broken up into small chunks. In a multi-processor or multi-core

system, the OS's thread scheduler dynamically puts them into the correct lane for most efficient processing.

Unfortunately, many applications, particularly those written for desktop or notebook PCs, aren't written with threading in mind. This is particularly problematic on desktops, where each application is designed to run in the foreground, and doesn't "play well with others." Take a typical desktop app: The entire application runs in a single thread. While the OS's thread scheduler will put it onto any available processor, the entire application is constrained to stay in that single thread. To horribly misuse my highway analogy, the application is a convoy that doesn't know how to change lanes, so while other cars are zipping all around it, the whole chain is constrained by the performance of its slowest components, like its I/O functions. So, if one driver has to stop for gas or hits a red light (i.e., is stuck in a wait state), the entire convoy has to stop.

In this case, however, the problem isn't that the thread is stuck behind some other application (the operating system will use thread scheduling to solve that problem). It's that in a single-threaded application, performance is constrained by the least-efficient part of that application itself. So while the app is reading data from the relatively slow disk drive, other meaningful processing within that app comes to a halt. While the Linux or Windows user- interface handlers do a good job of masking this, because they'll keep the mouse moving and so on, a single-threaded application can't do more than one thing at a time unless it's written to use threads.

Historically, of course, desktops and PCs could only process one thread at a time (i.e., they were on single lane highways), so the benefits of writing multi-threaded applications were minimal. But today, when it's clear that in a few years, most consumer and business desktops will be multi-core machines, it's time to begin finding the bottlenecks in those single-threaded apps, and rewriting them to handle threading. New applications should also be designed, right from the outset, to use threading.

Next Time: More Silliness?
In Part II, we'll dive into the issues of developing threaded applications. We'll discuss task-level parallelism, the implications of threading on compiled and managed languages, and introduce concepts like implicit parallelization and explicit parallelization of code using OpenMP. We'll conclude by talking about threading tools, and why you'll need them for analyzing and debugging multithreaded applications.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Driving in the Fast Lane: Multi-Core Computing for Programmers, Part 2

Learn how to leverage the capabilities of multiple cores to create more efficient applications. It's all about threading – multiple threading, that is.

by Alan Zeichick

Take a look at your computer. If you're running Linux or Solaris, check the list of processes, if you're using Windows, view the Task Manager. Your workstation is likely running dozens of processes. Many (or most) of those are owned by the operating system; today's modern operating systems are all extensively multithreaded. Some of those processes involve background tasks that are waiting, without consuming many CPU cycles most of the time. On my Windows desktop, there's an iPod service application, Norton AntiVirus, the Acrobat tray icon, and quite a few schedulers. There are also active applications, such as Word (currently in the foreground), AOL Instant Messenger (currently dormant), iTunes (playing "The Chain," by Fleetwood Mac), and Eudora, among others. There are also the core tasks that keep Windows alive—the network listener, UI handlers, and some device drivers.

A core part of any operating system is its task or process scheduler, which decides which threads and processes get access to a core, and how long they can stay there. Applications that spawn multiple threads, like Eudora, Word, and Outlook, provide the operating system more to work with. If Eudora or Outlook uses one thread to handle the user interface and another to communicate with the network, the OS helps it be more responsive to the user, and a better citizen when it comes to sharing resources.

Because modern operating systems are extensively multi-threaded, they automatically benefit any time you install them onto a multi-cored environment, where each core can take full advantage of system resources, each has its own caches, and each can directly access its own cache, main memory, and I/O.

In my particular case, my everyday workstation uses two single-core processors, but the effect is the same as if I had a single dual-core CPU. Windows XP automatically schedules tasks to run on both cores. Therefore, even if I don't do a thing with my applications--and even if my applications are not written to use threads at all—I benefit from having multiple cores at my disposal, because more threads can execute at one time, and the OS has more flexibility to maximize resource utilization.

For example, the operating system can keep a UI thread active and current even while a disk I/O thread is running, so that the UI doesn't have to halt momentarily while the OS preemptively swaps it out to service the disk I/O thread. Figure 1 shows the types of thread resources my dual-processor system is using at this particular moment.

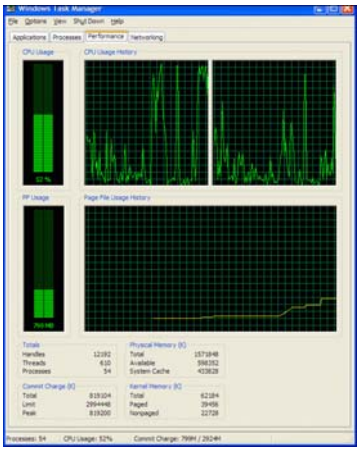

Figure 1. While Alan is writing this article, his dual-processor Opteron workstation is keeping both processors busy with 610 threads in 54 separate processes.

More Than the OS

If the only multithreaded software on your system consists of the operating system, you'll definitely benefit—the OS spins out hundreds of threads. However, you're leaving performance on the table, not to mention user.

Take Outlook. If it were a single-threaded application, it wouldn't let you write new messages while it was sending/receiving messages from the server. But because it's a well-threaded application, it can do both tasks. On a single-core system, the OS handles task scheduling, giving both the UI and send/receive threads some attention, but neither can run at the same time. In a multi-core system, the scheduler can let both run at the same time—or if network conditions slow, it can put a wait on the send/receive threads without affecting the UI threads at all. The result is an application that's more effective and more satisfying.

Many modern commercial applications (but not all of them) are written to use multiple threads. Sometimes that threading is explicitly designed into the app, like it was with Outlook. Sometimes the threads appear because the optimizing compiler implicitly threaded some parts of the application during the build process. That's great for commercial software, but the challenge is to bring that same level of threading into your own applications, whether they're being written for customers/commercial sale, or for in-house usage. That will not only make the applications better to use (think about the jerky editor on a single-threaded e-mail app), but also better able to exploit the performance of multi-core desktop and notebook PCs.

Note: Everything in this article applies equally well to server applications, but I'm going to focus on desktops/notebooks because enterprise server software developers have been creating threaded applications for many years. Because the only multi-cored PCs were rare dual-processor workstations and gamer systems, mainstream desktop app developers have had fewer reasons to explore threading. Because of this, but also because desktop users are more sensitive to the perceived performance of foreground tasks, the introduction of dual-core chips presents a more significant paradigm shift for desktop developers than it is for server app developers.

Three Approaches to Threading

When you look at threading applications, there are three ways to go, and they greatly depend on two separate questions: First, are you building from scratch or trying to retrofit an existing

application? Second, are you going to be compiling native code or are you writing for a managed runtime environment, such as .NET or Java? Depending on those answers, you can look at task-level parallelism, compiler-driven parallelism, or managed parallelism.

Task-Level Parallelism. If you're writing a new application, you have the opportunity to design the application in order to parallelize at the task layer. That's the best way to go; you can see that in applications like Outlook or Eudora, which are architected in a way that logically separates different tasks into different threads, and which use semaphores and other mechanisms to enable communication between those threads.

I'll be honest: Writing an application to exploit *task-level parallelism* can be difficult at first; it's like making the transition from structured to object-oriented programming. Think carefully about the best ways to split the code. Some obvious places are areas that have distinctly different needs. For example, a UI layer is I/O bound, but handwriting recognition is CPU-bound. In an application that uses network databases, one thread can communicate and sync with the network and fill a local cache, while another thread works out of that cache.

For another example, a game application might use task-level parallelism to separate out the game physics, the audio, the graphics, the strategy, and the different artificial intelligence characters in the game. If the game runs over a LAN, the related housekeeping and communication over the network could be two separate tasks.

The beauty of task-level parallelism is that it's independent of the underlying technology—it works well on both compiled and managed applications, and doesn't require special compilers. The downside is that it's difficult to retrofit a new design to existing applications, unless they're already written in a manner that happens to be sufficiently modular.

Compiler-Driven Parallelism. Independently of whether or not you implement task-level parallelism, you can—and should—enlist your compiler to help parallelize your native code.

Some optimizing compilers for C/C++ and Fortran can implicitly parallelize code by analyzing chunks of application logic. A typical optimization would be to split a loop into several pieces, each of which can be run in its own thread. For example, say you have a loop that iterates 1000 times in order to perform some operation on elements of a data structure, and that it's not important that those operations be performed sequentially. The compiler might split it into two separate threads, one with a loop that initializes elements 0-499, the other which takes care of elements 500-999. Clearly, there must not be loop-carried data dependencies that might cause the parallelization to mess up the results. If the compiler's not sure, it'll leave it alone.

Refer to your compiler's documentation to see what levels of implicit parallelization it offers. You'll generally need to invoke such parallelization through the use of switches. The best news about implicit parallelization is that it requires no source code changes.

A more powerful technique is to use *explicit parallelization*, such as through the use of OpenMP directive. OpenMP [http://www.openmp.org/] is an industry standard that allows you to easily indicate, inside the source code, which bits of code can and should be split off into different threads.

If you then compile using the appropriate switches, the compiler will dutifully generate the multithreaded code, taking care of all the necessary plumbing. If you don't specify the OpenMP compiler switches, the compiler ignores those directives, and generates single-threaded code. This lets you compile both ways, then perform regression tests and benchmarks to see if there were any unwanted effects from the parallelization.

This is another case when I'm going to say, "Refer to your compiler documentation for more information." Future articles on the AMD64 DevSource will discuss programming with OpenMP.

Managed Parallelism. Implicit parallelism, and explicit parallelism using OpenMP, are wonderful for adding threads to compiled code. But what about *managed* code for Java or .NET? The good news is that the Java Runtime Environment and the Common Language Runtime mask a lot of the details of application execution and optimization. The bad news is that, for the most part, there's not much you can do to drive increased threading. So, your best bet is to focus on task-level parallelization, and don't worry about using threads to improve the performance of individual tasks.

With that said, you can help improve the threading of specific tasks, but you'll have to handle all the plumbing yourself.

If you're coding in Java, use the Java Thread API to instantiate an object of type Thread and then send it a start() message to spin it off into its own thread. A good tutorial on this is on the IBM developerWorks site; see http://www-128.ibm.com/developerworks/library/j-thread.html.

What about .NET? Microsoft has provided a great deal of documentation on threading managed applications. See the.NET Developer's Guide [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconmanagedthreadingbasics.asp] for details.

You'll Also Need Tools

Let's say you've gone through all the steps above—you've implemented task-level parallelism, leveraged implicit and explicit threading with the C++ or Fortran compiler or using the threading facilities in Java and .NET. Now… How efficient is your threaded code? That's a good question. How about debugging threads? That's another good question. The answers typically require specialized tools.

A future article here on the AMD64 DevSource is going to look more deeply at the specialized threading tools, but I'll point you to one of them, the AMD **CodeAnalyst Performance Analyzer**. This is a neat tool for both 32-bit and 64-bit Linux and Windows—and it's free.

The Windows version provides a wide range of analysis, including system-wide profiling and thread utilization for both compiled and .NET applications. The Linux version, for the Red Hat and SUSE 2.4 or 2.6 kernel, can handle both native and Java applications.

And now if you'll excuse me, it's time to find some new music; iTunes has finished its Fleetwood Mac run. I think the appropriate next song for thinking about dual-core processors might be by Phil Collins: Anyone for "Two Hearts"?

Multiple Threads, Multiple Cores, Multiple Gains: Part 1

Learn how to evaluate the benefits of preemptive multitasking and the opportunities for utilizing it.

by Peter Aitken

When I used to work at a major university, my colleagues would sometimes complement me on my ability to multitask. I could spend a few minutes on the phone, then return immediately to writing an article, then stop doing that to talk to a student for a while, then read my mail, and so on. I suppose this was a useful skill, but that fact is that it was *not* multitasking—I was never actually doing more than one thing at the same time. Rather, what I was doing was more accurately called task switching, moving quickly and without interruption from one task to another. At the time I was unaware that my behavior modeled what goes on in just about every desktop computer.

We've all sat at our computer and seen two or more things going on at the same time. Who hasn't worked on a spreadsheet while another document is being printed and a large file is downloading from the Web? This sure seems like multitasking, but the fact is that the vast majority of computers can't multitask (in the strict sense of the word), all they can do is task switch—just like me.

While a central processing unit, or CPU—the "brains" of a computer—can do only one thing at a time, it can do things very quickly indeed. Modern operating systems such as Windows and the Mac OS have taken advantage of this fact to implement a form of task switching that is called *preemptive multitasking.* Let's see how this works.

Suppose that three programs are active—it doesn't matter what they are or what they are doing, we'll just call them A, B, and C. The OS lets program A use the CPU for some period of time, perhaps 10 milliseconds (the actual duration of the "time slots" depends on the CPU and OS in use). At the end of the time period the OS saves all the information about what A was doing and turns the CPU over to program B. After another 10 milliseconds, program C gets its 10 milliseconds, and then it's back to A again. Each time there is a task switch, the information about the task that's pausing is saved and the information about the task that is resuming is restored.

Because human perceptions are a lot slower than the CPU, and because most peripheral devices such as printers and networks are also slower, there's no perception on the user's part that anything has paused. Even if a program has only 10% of the CPU time, that is plenty to take care of the fastest typist and the speediest printer. What we see is a computer that is happily doing 2, 3, or more things at the same time. While the term multitasking may not be precisely correct for what a one-CPU computer does, the term is used in that way by most people and I shall do so also.

Preemptive Multitasking and Priorities

I mentioned that Windows supports preemptive multitasking. *Preemptive* means simply that the OS is in charge of what programs get CPU time, how much time they get, and when they get it. In other words, the OS can preempt a program in order to let another run. This is in contrast to *cooperative multitasking* where the programs themselves voluntarily relinquish the CPU after having executed for some period. Cooperative multitasking was used by the Windows OS prior to Windows 95 and Windows NT, and by the Macintosh prior to OS X. Cooperative multitasking makes OS design much simpler because the OS does not have to monitor and control program

use of the CPU, but it is less stable because a single badly written application can hog the CPU.

You might think that different programs would have different levels of need for the CPU—and you would be 100 percent correct. Windows defines four levels of priority:

- Real time - used for tasks that cannot be interrupted, such as streaming video and games with complex graphics.
- High - used for time-critical tasks that need to execute essentially immediately to function properly, such as the Windows Task manager.
- Normal - used for tasks that have no special CPU scheduling demands. Most application programs, such as a word processor or e-mail client, use this priority.
- Idle - used for processes that should run only when no higher priority process is active, in other words when the system is idle. A background backup or file indexing app would fall into this category.

Priorities are assigned by the OS. A program can request a particular priority and generally speaking will get it, but the OS has the final say.

Multithreading
You have probably heard the term *multithreading.* What is this, and how does it relate to multitasking? First, let's go over some terminology. A program running on a computer is technically known as a process. Each process has its own memory context—a region of the system's RAM that it and only it has access to. The OS prevents any process from meddling with memory that does not belong to it. The reason for this is probably obvious—if one program could alter memory belonging to another program, or to the OS itself, it is a recipe for disaster!

A *thread* is the basic unit to which the OS assigns processor time. By default, each program—that is, each process—has a single thread. As the OS gives each program its slot of CPU time it is actually assigning time to the process's thread.

Now we can understand multithreading. It refers to the case when a single process has more than one thread. Most programming languages provide developers with the tools to create and use additional threads in a process. When a process has more than one thread, all of the threads have access to the process's memory context. This makes it easy for threads to exchange data, but it also opens the door to potential problems that I'll discuss soon.

Multithreading Benefits
What's important to realize are the consequences of a process having two or more threads:

- Each thread will be allocated its own share of CPU time so the program as a whole can get more time than if it had just one thread.
- One busy thread—for example, one that is performing a long numerical calculation—will not prevent other parts of the program (that are on other threads) from running.

You might think that the first of these factors would result in a multithreaded program running faster. Generally speaking, however, that is not the case, at least not on single CPU systems. With multi-processor systems, as exemplified by AMD's multi-core designs, true performance gains are possible. Sometimes very significant performance gains can be had from multiple threads—more on this in the next article.

It's the second factor that motivates most use of multithreading in today's applications. If the program has to perform a CPU-intensive task, such as a long mathematical calculation, sorting a large amount of data, or loading a large file from disk, that task can fully occupy a non-multithreaded program's only thread.

And guess what? That lone thread is also responsible for responding to mouse and keyboard input and updating the display. The result is that while the time-consuming task is executing, the program "freezes" and does not respond to user input or update the display. This is a sure way to annoy your users, and most programmers know to put such time-consuming tasks on their own thread, leaving the program's default thread free to respond to user input.

What's Next?
Systems with more than one processor open the door to true multitasking, with two or more tasks running literally at the same time. Such true multitasking environments also enable multithreaded programs to do more than one of their own chores at the same time. As we'll see in the next two articles, multithreading can lead to significant performance gains, but at the same time make programming a more complex challenge for you.

Until recently, multiprocessor systems were designed by putting two or more discrete CPUs on a motherboard, and were generally limited to high-end servers and graphics workstations. Recently, AMD has introduced CPUs that include multiple processors, or cores, on a single chip. These designs are rapidly entering the mainstream, and are available on servers, desktops, and laptops today. Multi-core CPUs open the door to many possibilities but also pose unique programming challenges. We'll look at these possibilities and challenges in the next article.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Multiple Threads, Multiple Cores, Multiple Gains: Part 2

Learn how to use multiple cores for faster performance.

Peter Aitken

In the first article of this three-part series, we saw how modern operating systems, such as Windows, can rapidly switch a single CPU between different tasks such as printing a document, downloading a file, and performing mathematical calculations. Because this happens so quickly, the perception is that these tasks are happening simultaneously. We also saw how a program can be written to use more than one thread, so that a time-consuming task would not freeze the program's user interface. These techniques, multitasking and multi-threading, are today fundamental to the ways that we use our computers.

The fact is, however, that with the vast majority of systems, there is still only a single CPU doing all the work. How can we speed things up? The obvious route to better performance is to design computers with multiple CPUs—after all, it is a lot easier to add a second CPU to a computer than it is to make one CPU twice as fast!

Multi-CPU Computers
Systems with two or more CPUs are not new at all, and some approaches date as far back as the early mainframe days. In today's specialized world of supercomputers, designers have been linking larger and larger numbers of ordinary processors together to generate truly awesome computing speed.

For example, in 2005, IBM's Blue Gene/L used its 65,536 processors to break the existing speed record with an astounding 135.5 TFLOPS (FLOPS, or floating point operation per second, is a standard measure of computing speed, and a TFLOP, or TeraFLOP, is a trillion FLOPS—or 10^{12} FLOPS). More recently, the Lawrence Livermore National Laboratory announced that its 131,072-processor machine has reached 207 TFLOPS. In contrast, a fast personal computer might reach 10 GigaFLOPS (GFLOP = 10^9).

These supercomputers are all specialized for computation-intensive tasks such as weather forecasting, modeling nuclear weapons, and molecular simulations. Such models work in multiprocessor computers only because their tasks can be broken into many separate subtasks that can be performed in parallel. This, in turn, is possible only when each subtask does not depend, or depends very little, on the results of other subtasks in the model. For example, in weather forecasting, a large area can be conceptually divided into thousands of smaller square areas, and the weather calculations for each area can be assigned to one processor.

In the less exotic world of personal computers, two- and four-CPU systems, with each CPU on a discrete chip, have been around for a while. This technology is known as *symmetrical multi-processing*, or SMP, because all of the processors are identical to each other. These systems tended to be expensive and have been used primarily as servers and high-end graphics workstations, where the added throughput justified the extra cost.

The fact is, however, that most programs cannot currently benefit from multiple processors. You will remember from the first article that the operating system assigns CPU time to program threads, but most programs have only the single, default thread. The system could have a dozen

CPUs, and a single-threaded program would still not take advantage of them. There are exceptions, however. One example is Adobe Photoshop, which has supported multi-threading for quite awhile now and can take advantage of multiple processors when they are present.

Multi-CPU systems can offer benefits to the end user, even when he or she is not using multi-threaded apps. It's a rare system that does not have multiple things going on at the same time, even when only a single application program is running. A slew of background processes is often at work performing tasks such as system maintenance, virus scanning, and file indexing. When two or more CPUs are available, these tasks can run faster and more efficiently by being assigned to different threads—and therefore to different CPUs—by the operating system.

Multi-Core CPUs
As integrated circuit manufacturing techniques have improved, several factors have combined to motivate the development of multi-core chips that contain more than one CPU. Some of the more important factors were:

- It has become becoming increasingly difficult to increase performance by upping the processor clock speed.
- The use of existing, proven core designs in new configurations decreases development time, cost, and both financial and technical risk.
- Multi-core chips lower manufacturing costs. That is true for both the chips themselves, because it is cheaper to manufacture one *n*-core chip than *n* single core chips, and for the motherboards, because a multi-core chip requires less motherboard real estate, only one socket, and fewer interconnections.

On a more technical level, multi-core designs offered additional advantages over implementing SMP with multiple discrete processors. One advantage has to do with the cache coherency circuitry. Each CPU has a local cache of extremely fast memory where it keeps a copy of data from the system's main RAM. With multiple processors, it is possible for one processor to change the data in RAM, leaving another processor's cache with outdated data. Cache coherency circuitry monitors changes to RAM data and ensures that all processor caches have the most recent data. When processors, or cores, are on the same physical chip, the cache coherency circuitry can operate at the very high speeds of on-chip communication, rather than having to use the much slower between-chip communication, as is required with discrete processors on separate chips.

Another advantage of multi-core chips is related to power consumption. An *n*-core processor requires less power than the equivalent number of discrete single-core CPUs, partly because on a multi-core chip, the cores can share some circuitry, such as the HyperTransport interface, that must be duplicated on each single-core CPU, and partly because there is less external support circuitry required on the motherboard. Less power consumption means less heat to dissipate, longer laptop battery life, and lower energy costs.

From the outset of AMD’s 64-bit processor strategy, its architecture was designed to accommodate multi-cores on a single chip. Currently, AMD is producing dual-core processors for desktops, notebooks, and servers. Quad-core processors are slated for availability next year.

Software Design Considerations
For the vast majority of end users, working with a multi-core system is totally transparent. Some

programs may run faster, and background tasks will complete more quickly, but otherwise, you can just go about your work as you always have. For software developers, however, things are not quite so simple, because, just as with any SMP system, making the best use of the power of SMP can pose some thorny challenges.

Some challenges arise at the conceptual or design level. If a programmer wants to use multi-threading on a multi-processor system to truly increase performance, it is necessary to break a time-consuming task into two or more parts that can each be assigned to its own thread. Because these threads will be executing at the same time, each on its own processor, they need to be largely independent of each other, although not necessarily completely independent. Simply put, Thread A cannot be dependent on the results of Thread B to carry out its actions, and *vice versa*.

Some computing tasks are inherently linear in nature and will resist all attempts to divide them up for faster processing with multiple threads. Other tasks require a great deal of creative thought and experimentation before they can be successfully broken down for parallel processing. Other potential problems with multi-threading exist at the level of source code—or, as is often said, the devil is in the details.

One such problem is the so-called *race condition*, in which the outcome of the overall process is dependent on the order in which the individual parts, or threads, complete. Race conditions can occur when two or more threads access and change the same data, as often happens when multiple threads are not totally independent of each other.

For example, suppose that at some point during its execution Thread A changes the variable X, and at some point during its execution Thread B reads the variable X. The overall outcome will differ depending on whether Thread B reads variable X before or after Thread A changes it. Programmers must use various thread synchronization techniques to prevent such race conditions from invalidating their program's results.

A related potential problem is a *deadlock*. It occurs when two or more threads are each waiting for the other to finish something, with the result that all threads are frozen. As an example, consider an application that uses multiple threads to perform complex operations on a database. To avoid race conditions, each thread puts a lock on each individual table that it accesses and releases the lock when finished. Then the following could occur:

1. Thread A puts a lock on Table X.
2. Thread B puts a lock on Table Y.
3. Thread A now needs to access Table Y but cannot get it because that table is locked by Thread B (which is waiting). Hence, Thread A waits and never releases Table X which prevents Thread B from continuing
4. Thread B now needs to access Table X but cannot get it because that table is locked by Thread A (which is waiting). Hence, Thread B waits and never releases Table Y which prevents Thread A from continuing.

Avoiding deadlocks and race conditions is a difficult programming task, and a well-designed multi-threaded program will include not only the logic to prevent them but also code to resolve them when and if they do occur—something that not even the best programmer can always prevent.

Multi-Threading Today
Despite the difficulties that are inherent in writing multi-threaded programs, numerous companies have surmounted these challenges. In the third article in this series, we'll take a look at Modo, a state-of-the-art three-dimensional design and modeling package from Luxology that was written to take full advantage of modern multiprocessor systems. We'll also discuss how careful architecture of a program's design can optimize performance on a multi-core or other SMP system, and why careless architecture can lead to disastrous results.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

A Quick Look at the Benefits of Multi-Core Computing

Learn how AMD Direct Connect Architecture and AMD HyperTransport technology can help you improve your multi-tasking.

by Tyler Anderson

Have you ever tried to run multiple tasks on your current computer? Some days it slows to a crawl like it's trying to wait until it has dried out under the sun. And think of those excruciating freezes that can happen when you try to run the CD/DVD drive and an application at the same time.

That's where multi-core systems can help. The term sounds pretty cool—multi-core—but what does it mean? It means multiple processors combined onto a single silicon chip. This is not a small undertaking, and it involves engineering ingenuity. However the payoffs are great, with tremendous value for the home and business user.

I recall the first time that I ever used a multi-processor system. It was back in my days as a Research Assistant in an impoverished research lab at the university I attended. The reason I say impoverished is because the computer was a Pentium III (in a time when AMD had already released their dual-core Opteron and Athlon 64 processors and Intel had released Hyper-Threading). However, this was not just any Pentium III computer; it had two processors on it—the latest and greatest setup of its time (a decade ago)! Booting up and initial loading of your favorite and essential apps on even today's modern processors can be slow, but on this computer I could load several apps, yet I was always able to simultaneously play with the mouse cursor while I waited for Mozilla to load in the background. I don't know about you, but I was impressed. Being able to play with the mouse while waiting, or be able to comfortably compile source code and run compute-intensive applications at the same time? Now that's true multi-tasking.

There's a name for this—Multi-Processing (MP), which distributes loads across multiple processors in a single system. MP allows you to do more than play with a mouse—potentially a home user could burn a DVD or CD at the same time they are downloading music from the Internet. Though technically you can do those things on a single-processor system now thanks to time-sharing, the problem is that a single CPU can easily be overloaded. A user burning a DVD is likely to just create a coaster if there aren't enough CPU cycles available to meet the demand. Having more than one CPU makes this sort of multi-tasking much easier. In business and science, MP is a way to support multiple users and their applications on a single system. In the past, MP has most commonly been seen in very high-end home desktops, business and scientific workstations, and servers.

For the home user, multi-core means they'll now get the power of a workstation on their desktop, and they'll be able to do a lot more in terms of running more applications at the same time. So what does it mean for developers? More than you might expect.

The AMD64 Design
Why multi-core? Power consumption, heat, and their effects on clock speeds are a major factor in pushing the industry towards multi-core technology. At the current state of the art with AMD's manufacturing process, it's been found that going back two 200 MHz steps from the maximum clock frequency decreases power consumption by 40%. In other words, at high frequencies, the power consumption increases disproportionably as the clock frequency goes up. The most

important factor affecting power is the number of transistors, and that's affected by the micro-architecture and the size of on-chip cache (typically the L1 and L2 caches). All this adds up to current leakage and thermal issues that snowball with greater clock frequencies. Bottom line is that the clock cycle increases we've seen in the past can't be sustained, and that leads us to multi-core.

The trouble with adding all this extra computing power of multi-core is the load it will place on the system. Imagine a 4-way (aka 4-socket) system with dual-core CPUs; that's a total of 8 cores. In the past, most MP systems have traditionally been Symmetric Multi-Processing, in which multiple CPUs share a single main memory. With traditional SMP, it's too easy to stall the system by consuming all the memory bandwidth. If it's possible to overload a 4-way single-core SMP system, then 8 cores is going to just make things worse.

AMD planned for multi-core when it designed AMD64. The most obvious facts of AMD64 are 64-bit addressability and 16 64-bit registers instead of 8 32-bit registers. But there's a lot more to AMD64 besides that. AMD recognized the scalability problems inherent in traditional SMP designs, which is why you don't often see commodity servers with more than four processors (unless they're AMD64). CPUs on SMP can easily be starved because of the lack of memory bandwidth. Another subtle but real problem is cache coherency—correct functionality requires that CPU caches mirror each other if they address the same data; the synchronization overhead for this can bottleneck a system, severely impairing scalability. AMD improves scalability tremendously with the Direct Connect Architecture that allows for great scalability with multi-core. DCA is a smart alternative to traditional SMP.

Scalability and the Direct Connect Architecture

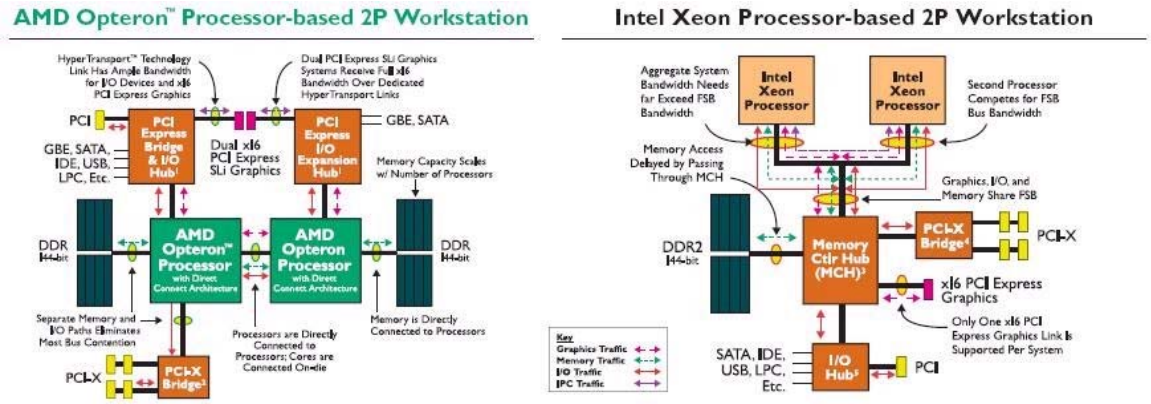

Figure 1. Key differences in the architecture connecting the AMD Opteron processor vs. the Intel Xeon.

One of the most important aspects of the AMD64 design is AMD's Direct Connect Architecture. Traditional SMP systems are bottlenecked by the Front Side Bus and memory controller hub, inhibiting top multi-core performance and capabilities. Picture the nasty highways on your way to work. In the single core architecture, we have two points, memory and other I/O at your place of residence, and the processor sitting where you work with a long, two lane highway connecting them. The traffic along the highway, instead of vehicles, is packets of data to and from memory

and I/O. Traffic can't get too bogged down since there is only one exit on each highway-at the end. Now imagine not one, but two processors side by side on the one side of the school using the same highway. What do you think will happen? Optimally, both processors won't be trying to send requests to memory at the same time or the highway will get more and more congested. Not only does a processor have to wait for data to come back from memory, but a processor also has to take turns sharing the highway with the other processor. Figure 1 shows key differences in the architecture connecting the AMD Opteron versus the Intel Xeon.

In AMD's Direct Connect Architecture, each multi-core chip gets its own memory controller, reducing the bottleneck point on traditional multi-core systems at the Northbridge where each processor would have to fight for a chance at memory. The old two-CPU Pentium III I used to work on suffered from this problem, too. Even though there were two processors in the computer, the maximum memory bandwidth didn't change, causing overall throughput to not increase as much as one would like or hope. Good thing the memory requirements for playing with the mouse pointer aren't too high.

With AMD's Direct Connect Architecture, the memory bottleneck problem is alleviated, since the DCA design is also a Non-Uniform Memory Architecture (NUMA) design for systems with two or more processors. In the scenario of two dual-core AMD64 processors, instead of one highway connecting both dual-core CPUs to memory, each CPU gets its own highway—*and* separate highways to both memory (via an integrated memory controller) and I/O. In a nutshell, AMD's Direct Connect Architecture connects processors directly to each other, memory, and I/O. This greatly reduces the bottleneck, resulting in increased performance.

AMD's HyperTransport Technology is the booster that adds speed to the highways in the Direct Connect Architecture. Thus, instead of sluggish traffic-prone highways, something more like speedways (or several-lane highways) exists, three of them in fact, apart from the integrated memory controller. Each of the three links are known as HyperTransport links with up to 8 GBs worth of bandwidth per CPU with the memory controller providing up to an additional 6.4 GBs, allowing a top speed of roughly 30 GBs. Take a look at Figure 2 for comparison data with AMD Opteron versus Intel Xeon, Pentium 4, and Apple's G5.

Workstation System Comparison	AMD Opteron™	Intel Xeon[1]	Intel Xeon[2]	Intel Pentium® 4[3]	Apple G5[4]
Modular, glueless scalability	Yes	Requires Northbridge	Requires Northbridge	Requires Northbridge	Requires Northbridge
SMP Capabilities	Up to 8-way	Up to 2-way	Up to 2-way	1-way	Up to 2-way
Direct Connect Architecture	Yes	No	No	No	No
Dual-Core technology	Yes	No	No	No	No
High Performance 32-bit and 64-bit computing	AMD64	EM64T	EM64T	EM64T	Yes
HyperTransport™ technology	Yes	No	No	No	Yes
Integrated DDR memory controller	Yes	No	No	No	No
Front Side Bus frequency	1.4 – 2.6GHz[†]	533MHz	800MHz	800MHz	900MHz – 1.25GHz
Front Side Bus bandwidth	11.2 – 20.8GB/s[†]	4.3GB/s	6.4GB/s	6.4GB/s	7.2 – 10GB/s
Maximum Inter-processor bandwidth	8.0GB/s	4.3GB/s	6.4GB/s	N/A	7.2 – 10GB/s
Memory support	DDR200/266/333/400	DDR266	DDR333/DDR2-400	DDR2-400/533	DDR333/400
Memory Bandwidth IP System	6.4GB/s	4.3GB/s	6.4GB/s	8.5GB/s	6.4GB/s
Memory Bandwidth 2P System	12.8GB/s[††]	4.3GB/s	6.4GB/s	N/A	6.4GB/s
Memory Bandwidth 4P System	25.6GB/s[†††]	N/A	N/A	N/A	N/A
Maximum Graphics Support	x16 PCI-E (x16 SLi)	8X AGP	x16 PCI-E (x8 SLi)	x16 PCI-E (x8 SLi)	8X AGP
LI cache size (max.)	64KB (Data) + 64KB (Instruction) per core	8KB (Data) + 12k mop (Instruction)	16KB (Data) + 12k mop (Instruction)	16KB (Data) + 12k mop (Instruction)	32KB (Data) + 64KB (Instruction)
L2 cache size (max.)	1MB per core	512KB	2MB	2MB	512KB
L3 cache size (max.)	N/A	2MB	N/A	N/A	N/A
Maximum I/O bandwidth IP System	8.0GB/s[††]	3.2GB/s	4.3GB/s	2GB/s	4.3GB/s
Maximum I/O bandwidth 2P System	24.0GB/s[†††]	3.2GB/s	4.3GB/s	N/A	4.3GB/s
Maximum I/O bandwidth 4P System	32.0GB/s[††††]	N/A	N/A	N/A	N/A
SIMD Instruction Set Support	SSE, SSE2, SSE3	SSE, SSE2	SSE, SSE2, SSE3	SSE, SSE2, SSE3	AltiVec

Dedicated Bandwidth
Shared Bandwidth

[†]The front side bus (interface to memory) of the AMD Opteron™ processor runs at the speed of the processor
[††]AMD 1P System – AMD Opteron 100 Series with NVIDIA nForce Professional 2200 chipset
[†††]AMD 2P System – AMD Opteron 200 Series with 1 HyperTransport Inter-processor Bus and 3 HyperTransport I/O Buses with DDR400 memory
[††††]AMD 4P System – AMD Opteron 800 Series with 4 HyperTransport Inter-processor Buses and 4 HyperTransport I/O Buses with DDR400 memory

[1]With Intel E7505 chipset (http://developer.intel.com/products/chipsets/e7505/index.htm)
[2]With Intel E7525 chipset (http://developer.intel.com/products/chipsets/e7525/index.htm)
[3]With Intel 925X chipset (http://developer.intel.com/products/chipset/925x/index.htm)
[4]G5 processor in Apple Power Mac G5 workstation

Figure 2. Comparison data for an AMD Opteron processor-based workstation vs. Intel Xeon, Pentium 4, and Apple G5.

Can you see the scalability advantages as well? Adding a third CPU to the setup doesn't increase the load on the memory controllers of the first two CPUs, since each CPU gets its own integrated memory controller, its own connections to one or two neighboring CPUs, and I/O. With four CPUs in the mix, the four CPUs are positioned in a square. You can compare an Intel 2-socket system versus the AMD 2-socket system back in Figure 1. The 4-socket AMD system is shown in Figure 3. Positioned in this fashion, each processor can be connected to both of its closest neighboring CPUs via two HyperTransport links, memory via the memory controller, and/or I/O via the third HyperTransport link. The performance of each single CPU is no longer dependent on the other CPUs, thanks to the Direct Connect Architecture and HyperTransport Technology. We actually aren't limited to four CPUs; Direct Connect Architecture is designed for 8 CPUs, which means you can get an 8-way Opteron system as a highly scalable off-the-shelf, commodity-priced yet enterprise- capable server.

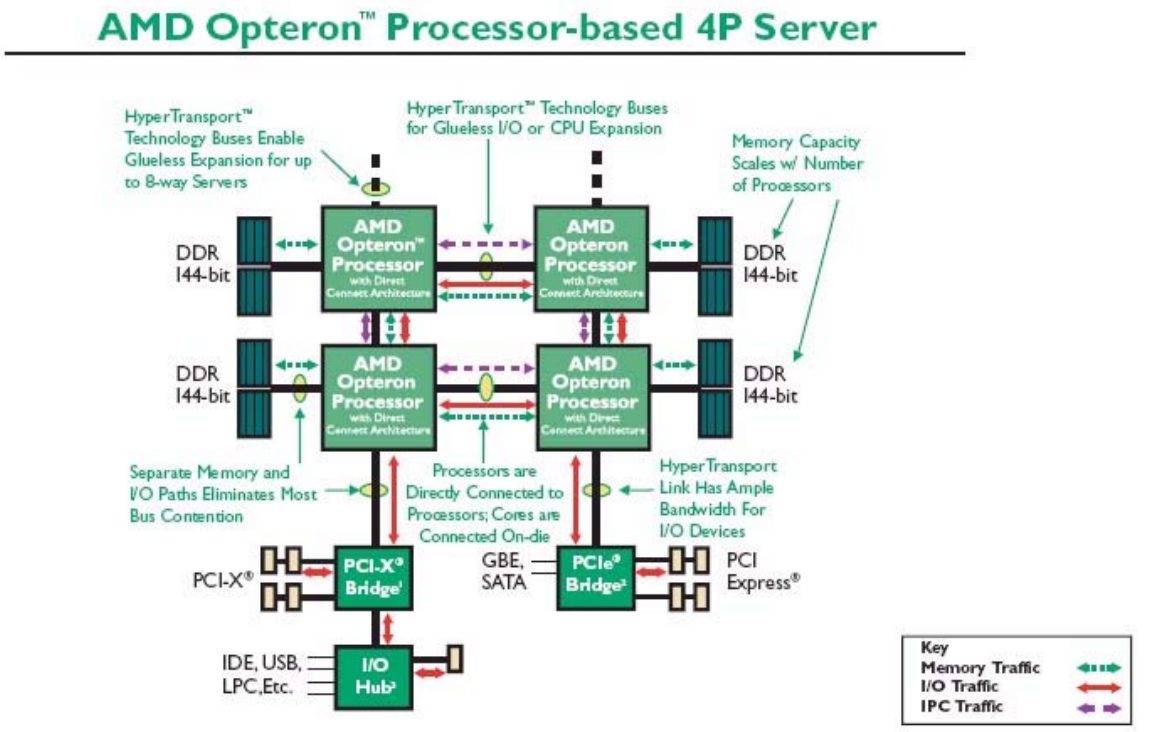

Figure 3. Diagram of an AMD Opteron processor-based 4P Server

Also remember the issue of cache coherency; with Direct Connect Architecture, the point-to-point HyperTransport links allow very fast cache synchronization. However on a traditional SMP system, cache synchronization is bottlenecked by the Front Side Bus and the memory controller hub.

Multi-Threading Concerns

Are we faster yet with multi-core? Maybe not. It's critical to understand whether applications are multi-threaded. If your application is not multi-threaded, it will not show any performance improvement with multi-core, other than the side effects you get from background processing such as downloading file, printing, or virus scans.

To really take advantage of multi-core, you need multi-threading. Because of all the previous history with SMP, there's already a lot of software out there that does this. With data parallel threading, units of data can be independently processed by the same piece of code, and this can be seen in applications such as digital signal processing and simulations. Another type of threading is task parallel threading which is different complex functions running asynchronously on different tasks; an example would be a web server handling JSP and CGI services.

Multi-threading smart with multi-core may take some thinking. Suppose for example that you have a number-crunching scientific application that uses data parallel threading and by default grabs every CPU when it multi-threads. Let's suppose it scales very well and it runs a job in 1/6th the time on 8 single-core CPUs versus one single-core CPU (scaling better than that is really hard to do). So it takes an hour with 1 CPU and 10 minutes with 8 CPUs. Throw dual-core into the mix,

giving 16 cores total. At best it may run in 5 minutes, though in reality it would probably be a little longer than that. The law of diminishing returns means that you may not care much whether it takes 5 or 10 minutes. Certainly, either one is much better than an hour. If the application had a tuning knob to allow it to just take 8 cores for the job, then that means there will be 8 cores available for other tasks or other users. To make this happen though, developers need to learn how to query the number of cores and provide tuning knobs for their users. OS vendors such as Microsoft are providing APIs for this purpose.

AMD is preparing for the future by laying the groundwork with the Direct Connect Architecture and HyperTransport Technology found in AMD64 single- and multi-core processors. Now that the technology is developed and proven in silicon-and as the demand for more memory bandwidth and more processors per system grows-AMD is already ahead of the game. Home users will get more power to let them do more than ever before. Developers that are interested in multi-threading need to find out about APIs for querying the core count in their favorite operating system. They also need to learn about data parallel versus task parallel threading techniques. Rest assured that AMD will continue to leap ahead of the competition in architectural strides, building upon this framework of technologies, with further revs of their processor lineups.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

To Thread, or Not to Thread, in a Virtualized World

Learn how threading and virtualization can work together to make your application fly.

Steve Apiki

It used to be, as application designers, we could follow processor news from a distance. Whatever changed underneath, our program's performance could always hitch a ride on increasing clock frequencies, or on the work of compiler writers and systems programmers. But multi-core processors like the Dual-Core AMD Opteron have changed the game, requiring that application designers more directly participate in the process. The only way to make the most of a Dual-Core AMD Opteron, or any future multi-core processor, is going to be to build in concurrency at the application level.

The latest processor news is all about virtualization. AMD-V virtualization technology, due later this year in the Opteron and Athlon 64 lines, delivers hardware support to virtual machine managers. The industry's shift to virtual machines, while maybe not as profound to application programmers as the dawn of multi-core, also bears watching because it changes the execution environment of our software.

How do these two factors, together, change the way we approach application design? At a high level, multi-core and virtualization seem to play against one another. That is, application designers could rewrite to support parallel execution on several cores, or they could, instead, anticipate programs that run isolated on a single virtual core, no matter what the underlying hardware. In the second case, the effort to carefully decompose the application into component threads might go largely unrewarded.

For many domains, especially on the client, finding tasks that can be handled in parallel isn't easy. And even when sections of code ripe for threading are obvious, implementing threads with today's tools is tough. Debugging is a third challenge. In short, building a heavily-threaded application is generally a more expensive proposition than building one that is single-threaded. So while multi-threaded applications are necessary to take full advantage of multi-core processors, there is a cost associated with the process. Not every application is going to justify that cost.

Virtualization on the Server: Scaling Out

Virtualization is about isolation—isolation so that different operating systems and their associated binaries can share the same hardware, isolation so that development builds can be tested safely, isolation for secure execution of untrusted code—but it is as also about effective resource utilization. Virtualization makes it possible to flexibly define sets of resources. For example, memory can be parceled out to different processes in different virtual machines, even overcommitted, and the allocations between these machines modified to adapt to changing loads or requirements.

Cores can be allocated to virtual machines in much the same way, at least under hypervisors that support SMP. These include VMware with Virtual SMP, and Xen as of version 3.0. Microsoft has also talked about supporting SMP in future releases. In these environments, multiple cores (or processors) can be assigned to single virtual machines. Hypervisor SMP support takes away the disincentive to thread applications that will likely run on virtual servers, because even here, different threads from the same application can be scheduled on different physical cores.

So, virtualization doesn't restrict you from gaining performance by threading applications on multi-core processors. But it does point toward another path to concurrency: scaling out with virtual clusters. Clusters are interesting in this context not so much because they are any easier to implement than threads, which they aren't, but because once built on a set of virtual nodes, the same design can be scaled literally outside of the box to a number of real nodes. A dual-core system running a set of virtual machines makes an attractive development and initial deployment cluster.

Concurrency Continuum

Clustering occupies the end of the concurrency scale that's farthest away from threads, but it is still concurrency, and concurrency is what is required to take full advantage of multi-core. Somewhere in the middle is the notion of partitioning an application by process, at a higher level than threads, but without the isolation and overhead of virtualization.

Having a Web-facing server communicating with a database back end isn't quite partitioning by process in the sense that we're discussing here. In this common arrangement, concurrency is actually achieved by threading within each process. The idea of process-level concurrency is to write single-threaded processes that can run in parallel, communicating with one another at the process level.

Processes are heavier than threads, of course, and with that weight comes advantages and disadvantages. The biggest advantage is that processes don't share memory, so the very real difficulties of managing shared memory access with locks and related synchronization objects disappear. The biggest disadvantage is that multiple processes don't share memory, so any kind of low-level cooperation involving shared data is out of the question.

On the server, the requirement to smoothly provide services to more than one client almost certainly makes concurrency already a design goal. Multi-core doesn't change that. Although threading is the prevailing mode, it may be helpful to keep in mind that there is a continuum of concurrency options, ranging from lightweight threads running in a shared memory space to a clustered server partitioned into heavyweight virtual machines.

AMD's dual-core design makes the Opteron well suited for any of these concurrency choices. I've been using the terms "core" and "processor" almost interchangeably, but dual-core doesn't quite equal dual-processor. A Dual-Core AMD Opteron shares a single memory controller and set of HyperTransport interfaces in common between its two cores. That limits each core's access to memory and I/O relative to a dual-processor Opteron system. Fortunately, the sharing of these resources is handled by an efficient crossbar switch, minimizing any performance loss. This architecture, along with hardware virtualization support, will make the Dual-Core AMD Opteron an excellent virtual machine platform.

On the other hand, multi-threaded designs might additionally exploit the dual-core Opteron's superbly efficient handling of shared memory. On a dual-core processor, threads from a single process can be scheduled on the same chip, on different cores. Shared data structures are likely in both L2 caches, and the Opteron manages cache coherency. Updates from one L2 cache to the other to ensure cache coherency run on-chip, through the system request interface. That means your multi-threaded application gets the benefit of running on multiple physical cores with trips to main memory kept to an absolute minimum.

Threading may be used in network servers as a means of dealing with I/O requests. An alternative network server technique, called select-based, uses just a single thread to multiplex network traffic. Select-based concurrency uses an operating system select() (or a related system call, depending on the specifics of the design) to monitor activity on a set of non-blocking sockets. This approach has proven to scale effectively, but multi-core might start to tip the balance back toward a multithreaded/blocking socket approach.

Threads on the Client

On the server, your choice between threading and not threading revolves around choosing alternative methods of providing concurrency. It's a different matter on the client, where concurrency is optional, but threading is the only practical concurrency technique. On the client, the question is simpler: is it worth the cost to thread this application, or not?

If you don't thread, your application may still see some benefit from dual-core, in the form of another core on which to run other applications, OS services, and background processes, such as anti-virus software. That may well be good enough for now. But the user's perception of "good enough" will change if other desktop software becomes aggressively multithreaded, and could change even further if a four-core Athlon 64 should materialize.

Virtualization doesn't change the client picture much, because it's unlikely that virtualization on the client will become so dominant a model that we could afford to ignore clients running on bare multi-core machines.

So unless you can accept a performance plateau in the near future, you will need to thread thick client applications. The short-term goal should be to identify the most obvious segments of your application that can benefit from concurrency, and to go after this low-hanging fruit.

On this list would be tasks long associated with parallel processing, such as rendering, speech recognition, and other media-related tasks. In some cases, these may already be handled by threaded code.

Actually implementing threads is by no means easy. Minimizing critical sections to keep threads running as freely as possible, avoiding deadlocks, and debugging subtle race conditions all come with the territory.

But it's after those obviously threadable sections have been built that we come to the really hard part, and that is factoring a generic desktop application into multiple threads. This is the stickiest part of the threading problem, and it's where we find ourselves right now, or will at the next design cycle. We're used to factoring designs along other axes: into classes, into modules, into model, view and controller. We're going to need to develop the skills to see opportunities for parallelization just as well, if we are to continue to bring client software along the performance curve.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Living in a Multi-Core World: Tips for Developers

Learn how to best take advantage of advances in multi-core processing.

by Justin Whitney

Multi-core has arrived. While multi-processor configurations have been around for awhile, AMD produced the world's first x86 multi-core processor in late 2005 with its Opteron 64 Dual-Core processor. Built primarily for servers, this paved the way for the Athlon 64 X2 Dual-Core, for desktops and laptops, and the Turion 64 X2 Dual-Core, created primarily for lightweight laptops.

As the first dual-core machines hit the market, consumers begin to discover the joys of serious multi-tasking, like listening to music while downloading a movie (legally), browsing, and running a virus scan, just as an example. But while multi-core processors give consumers major performance gains by virtue of their architecture, most developers still have a ways to go in learning to adapt to, and thrive in, a multi-core world.

Multi-Core Evolution
When AMD first put designs on the drawing board for a new architecture, it started with several parameters:

- Ease migration to multi-core, as well as heavy power consumption, by speccing the processor to existing footprints.
- Loosen bottlenecks inherent in sharing chip components between multiple CPUs.
- Fit well in a virtualized environment.
- Set the foundation for an evolution beyond dual-core, into quad-core and beyond.

To overcome their various hurdles, AMD implemented a number of new features, far too many to list here. But the features that are perhaps most pertinent to developers include:

- Improvements in power consumption
- Individual L2 cache per core
- Memory- and thread-affinity
- Positioning for virtualization

In addition to being familiar with these aspects of 64-bit and multi-core architecture, developers can also benefit from knowing what Microsoft has in store for a multi-core world with the upcoming release of Vista. That's covered here, as well as some specific developer tips for coding to multi-core platforms, both in managed and native code.

Staying Cool Under Pressure
AMD designed its latest dual-core processors to drop into existing 940-pin sockets (for AMD Opteron processors) and 939-pin sockets (for AMD Athlon 64 processors) that are compatible with 90nm single-core processor architectures. Not only does this allow IT managers to pop in new chips into old machines, assuming they upgrade the BIOS along the way, but it also helps with power consumption by not increasing physical space requirements.

AMD 64-bit processors use a power management mechanism called AMD PowerNow! Technology, also known as Cool 'n Quiet for the laptop set (it's the same technology, with different marketing terms for different consumers). Essentially, power use steps up or down based on processor load. At the lowest level, or "P5" in PowerNow! terminology, this means a power consumption of around 32 watts. At its highest level, or "P0", it's around 95 watts. This can mean a power reduction of 75 percent while Idle.

All of this exists in the single-core versions of Opteron and Athlon processors, so nothing new has been added specifically for the multi-core architecture. But because of these power management technologies, AMD is able to add more cores, and more processing power, without melting your desk.

What This Means to Developers
Simply put, you'll be coding for multi-core. If not now, then in the very near future. And in order for your applications to perform competitively, they'll need to make the most of the architecture. Not only that, but it also means planning for the stage. The multi-core architecture is designed to grow. Today, it's dual-core. Tomorrow, quad-core. After that, who knows? (Keep reading for tips on just how to take advantage of multiple cores beyond the first two.)

Benefits of a Separate L2 Cache
One of the potential bottlenecks in a dual-core configuration comes from the L2 cache. In a single-cache configuration, you get a performance hit when multiple threads are competing over the same data cache. Having a separate L2 cache for each core gives you twice the cache benefit. And of course, if you have four cores, each with its own cache, that's four times the benefit.

Having these dual L2 caches gives AMD's 64-bit architecture, also known as Direct Connect Architecture (DCA) one of its key distinctions over its competitors. But simply having this architecture in place only takes you so far. To really benefit from L2 cache separation, developers need to implement threading techniques that allow separate cores to process separate data sets, limiting cache contention and coherency problems.

For example, consider "functional threading": the first thread handles one distinct process, then passes the operation to the second thread in the pipeline, which is dependent on that data... then to the third, the fourth, and so on. While these operations can try to run in parallel, ultimately gains are limited because of contention over the same data cache.

But with "data parallel threading," you would create threads that rely on independent data sets, for example dividing a video frame into two halves. This allows concurrent threads to make full use of an individuated cache-core configuration. Also, coding your apps with an emphasis on parallel threading allows you to automatically scale up as processors begin to add even more cores to the die.

What This Means to Developers
A key strategy for living in a multi-core world is coding toward future processors. AMD has already announced a platform for gamers codenamed "4x4", which consists of two dual-cores, or four CPU cores (two sockets), and four GPU cores. The platform is designed to be upgradable to the new quad-core processor, to be launched in 2007, with a total of eight, count 'em eight, processor cores. Suffice it to say that threading will become an important part of your code, if it's not already. But how you thread will be just as important.

What Is NUMA?
Along the lines of an independent L2 cache, the AMD64 architecture also employs NUMA, Non-Uniform Memory Access (or Architecture, depending on who you ask). In this scenario, each processor socket has its own memory controller, shared by all cores on that processor, which is typically populated by the system with actual physical memory. For AMD, this especially becomes important when a configuration consists of multiple multi-core processors.

For each core, some memory is directly attached, yielding a lower latency, while some is not directly attached and has a resulting higher latency. When a given thread begins processing, the OS looks at which core is running the thread and allocates physical memory to that process. This way, the data stays close to the thread that needs it, a process called "memory affinity."

The OS considers this core to be the "home processor" for the thread and tries to keep the thread running on it. This "thread affinity" or "process affinity" contributes to performance by keeping the thread from unnecessarily getting moved. Each time the thread moves over to another core, performance takes a slight hit.

Considering that memory is the source of the most data traffic on the computer, even more than IO, this setup increases memory bandwidth at a ratio effectively the same as the number of cores. So a 4-socket server will have 4 times the memory bandwidth.

What This Means for Developers
If writing native code, do a memory allocation request for each thread. The OS will see this and handle the allocation by assigning memory in the physical bank attached to the processor on which that thread is running.

What's This About Virtualization?
In a typical server room, you have different machines being used at different levels of their capacity. Some apps take up more of a load, or may peak at certain times of the day. In a virtualized environment, hardware gets pooled so that resources can be shared and loads distributed in such a way that the same workload requires less equipment.

As a result, you have a scenario in which multiple instances of an OS are running across multiple machines, but each instance thinks it's the only one in that environment. A hypervisor keeps up the illusion and is the only bit of software in the space that really "knows" what's going on. Simply by virtue of its additional capacity, a multi-core processor fits perfectly into a virtualized environment. But AMD goes a step further by integrating virtualization assistance, previously codenamed Pacifica, further supporting applications that have been coded for such an environment.

What This Means for Developers
Huge heads-up for developers here... Because the hypervisor tricks the OS into seeing the hardware differently from what it really is, it would be dangerous to use low-level direct hardware access to detect processor characteristics and features. For example, don't use CPU ID instructions to get the number of cores on the machine. Instead, rely on the OS. After all, if the OS is fooled, everyone is fooled, but at least they all agree.

Something Vista This Way Comes
A big question developers have when it comes to multi-core is, "How does Vista change the

game?" Well, good news: Microsoft's latest major OS overhaul (with a marketing budget of $500 million) has been engineered to take advantage of both 64-bit and multi-core computing.

NUMA and Virtualization
In addition to running on processors with multiple cores, servers will continue migrating to multiple processors with multiple cores. In a system where each physical processor has its own memory, it's up to the OS to handle both memory- and thread-affinity. Microsoft, which expects all processors to be multi-core by the year 2009, has been giving these issues extra attention in Vista. Likewise, after it ships, Longhorn will get a free add-on called Windows Server Virtualization, with its own low-level hypervisor technology.

Visual Studio Compile Options
Visual Studio developers coding to .NET 3.0 on the Vista platform are expected to have the option of compiling directly to a specific processor. One of those options will be the AMD Opteron processor. What's the advantage? Normally, you compile your apps to an Intermediate Language (IL). Then, the first time for each session that the app executes on its deployment platform, the Just-In Time (JIT) compiler re-compiles the IL to that specific platform. This can mean a major performance hit when the JIT is taking place.

Compiling specifically to a particular processor, such as an AMD Opteron processor, alleviates that hit. Of course you'd have to be absolutely positive that the app will be running on that processor. If that's the case, however, you'll be able to take advantage of the AMD64 architecture out of the gate, even without additional code tweaks to implement multi-threading and other multi-core tricks.

Coding for Multi-Core: Basic Tips for Developers
So what exactly are some of these tricks? What can developers do to take advantage of the tremendous potential of multi-core? Well, short answer: it depends on what kind of developer you are.

Tips for Native Code
For low-level coders, here are the Top 5 Optimizations, as summarized from the Software Optimization Guide for AMD64 Processors, available from AMD Dev Central:

1. Memory Size matches: When storing and loading data, keep their operands aligned and match the sizes of the load/store.
2. Natural Alignment of Data Objects: locate your object at an address, which is a multiple of its size, for example locate a Word at an address evenly divisible by 2, or a Quadword at an address evenly divisible by 8.
3. Memory Copy: use the memcpy() function included with the Microsoft or gcc tools, which optimizes for all block sizes and alignments.
4. Branch Density: align branches so that they don't cross the 16-bit boundary.
5. Prefetch Instructions: use one of the prefetch instructions to take advantage of the high bus bandwidth and L2 cache loading on Opteron and Athlon 64 processors.

Tips for Managed Code
Far and away the best bet for .NET Developers is to focus on multi-threading, specifically using some of the techniques mentioned above. Use the various Threading objects available on the .NET Framework, such as:

- System.Threading Namespace
- Thread Class
- Threadpool Class
- Threadstart Delegate

For a demonstration of threading for multi-core, with before and after snapshots using AMD CodeAnalyst, read "Optimizing for Multi-Core with AMD CodeAnalyst."

To summarize, here are some tips for developers who want to take advantage of multi-core processor power:

- **Parallel Threading:** whenever possible, create threads that use independent data caches, rather than causing each thread to thrash the other's cache by relying on interdependent data pools.
- **Thread Affinity:** do a memory allocation request for each thread so that the OS can assign a memory controller and physical memory to that thread, in addition to assigning it a home base of its own core.
- **Trust the OS:** anyone who's run a beta version of Vista may cringe at this statement, but this actually refers to the OS's perception of its environment. Assume virtualization and avoid making low-level CPU queries. Instead, ask the OS what it thinks it's running on.
- **Most importantly**, anticipate more cores than you could ever anticipate. Dual-Core is just the beginning. As processors rapidly evolve, chances are high that your application will outlive the platform it originally deploys on. Prepare now and architect your apps to run on an X-Core processor grid. No telling what the world will look like tomorrow.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Implicit Threading, Explicit Threading: What's Best, How to Choose

Learn how to use OpenMP for simple, portable means of threading code and maximizing performance on multiple cores.

Anderson Bailey

AMD's multicore architecture lets you run multiple threads of execution simultaneously—each enjoying the full resources of a full processor core. Developers of desktop applications, however, face the task of rewriting code to exploit this parallel capability and of adopting a somewhat different view of program design and architecture.

Modern languages, such as Java, C#, VB.NET, Ruby, that rely on execution environments offer simplified interfaces to native threading resources, so that developers can quickly perform basic threading tasks.

However, in languages such as C/C++ and Fortran that compile directly to native code, the interfaces map to the native, low-level threading resources of the operating system. (Windows threads on Windows, Posix threads on Linux and most versions of UNIX). This aspect can make it more challenging to implement threading in these languages. In addition, the code that results is not likely to be portable.

Several years ago, a group of vendors came together to address the issues of complexity and the lack of portability of threaded code. They formed the OpenMP alliance (www.openmp.org) and delivered a standard for C/C++ and Fortran—called OpenMP—that is both portable and greatly simplifies the implementation of parallel code using implicit threading. This article examines the OpenMP approach and compares C/C++ code written for OpenMP to that of native Windows threading interfaces that is written to use explicit threading. Sometimes OpenMP is the best tool for the job. In fact, often it's the best tool. But not always, and you have to use an explicit approach. Here's the skinny.

Basic OpenMP

OpenMP consists of a set of pragmas, some APIs, environmental variables, and a few header files. In its most common use, only the pragmas are employed. In fact, the pragmas are the preferred way of using OpenMP.

Pragmas are compiler directives that are place directly in code. They have the special benefit that if the compiler does not recognize the pragma, the directive is simply ignored. Here is an example from OpenMP:

```
int i;
```


```
#pragma omp parallel for
for ( i = 0; i < ARRAY_SIZE; i++);
    array[i] += i;
```

The pragma, located just before the outer-most for-loop, tells the compiler to parallelize the loop. On a compiler that does not support OpenMP, this command is ignored and the loop would be compiled normally—as a single-threaded operation.

However, if the compiler does support OpenMP, it generates a lot of threading code in the background. Essentially, it will insert code that determines the best number of threads to use on the execution platform and then break up the loop across that number of threads. So, for example, on a dual-core system, it would break up this loop over two threads: one thread would handle elements 0 through ARRAY_SIZE/2, the second thread, running on the other core, would handle the remaining elements. If there were four available cores, the loop would be broken up over four threads—each handling one quarter of the elements in this array.

At the end of the for-loop, the code returns to single threaded until another OpenMP pragma is encountered.

Notice that this pragma has removed a lot of overhead and housekeeping, such as:

- Determining the number of threads to use
- Creating and starting the threads
- Breaking up the array into threadable sections
- Assigning different sections to each thread
- Synchronizing and terminating the threads

As we shall see shortly, native code requires all these steps be hand-coded by the developer.

Many programs derive the bulk of the advantage of parallelization in loop constructs, so this simple pragma can get programs much of the advantage of multicore.

OpenMP vs Native Windows Interfaces

The accompanying listings provide two versions of the same program—one using Windows native threads, the other OpenMP. The program is designed to give a very rough measure of the randomness of numbers. It reads a file of 10 million digits (included in the download at {link here} and stores it in memory. It then reads through that data converting every four digits into a short integer whose values range from 0 to 9999. Each integer is placed into an array of 2.5 million shorts.

The array is then traversed to see how frequently each of the possible 10,000 values occurs. (In a perfectly smooth distribution of digits, each value would occur 250 times.) A final count steps through the 10,000 values and computes the average deviation from 250. (Statistics cognoscenti will recognize that the sum of squares method should be used rather than average deviation, but the purpose of this exercise is primarily to exercise loops rather than write a true statistical analysis.)

Using OpenMP, this entire process can be done in 88 commented lines of C/C++ code. The key loop (which reads through the array of 10 million digits and converts them to shorts) is at lines 54-62:

```
#pragma omp parallel for
    for ( m = 0; m < DIGITS_ARRAY_SIZE; m += 4 )
    {
        short_array[m/4] =
            ( digits_array[m]   - '0' ) * 1000 +
            ( digits_array[m+1] - '0' ) * 100 +
            ( digits_array[m+2] - '0' ) * 10 +
              digits_array[m+3] - '0';
    }
```

The Windows native approach uses 152 lines of code. It requires a guess as to the best number of threads (in this case, four is chosen for a dual Opteron system with dual cores). The necessary thread structures are created (lines 23-33) and initialized (lines 69-80); the above loop is written as a separate thread function (lines 130-153); and this function is called four times—once for each thread (lines 81-97). Then the threading code must wait until all the threads finish before continuing on (lines 99-105). Clearly, OpenMP is simpler.

To use OpenMP, you need a compiler that supports it. As of late 2005, Microsoft's C/C++ compiler in Visual Studio .NET 2005 supports OpenMP. To use it, simply include OpenMP support in project properties and include the omp.h header file. You must also remember to include the OpenMP runtime, vcomp.dll, in your binary distribution files. Many other C/C++ and Fortran compilers support OpenMP. The benefit of this code, of course, is that it is portable across all these compilers, which is not true of the Windows native APIs, of course.

Not Gotcha-Free

The pragma shown above applies only to the immediately following for-loop. The code returns to single threaded as soon as the compiler encounters the closing brace of the for loop. Variables, such as m, declared in the loop are instantiated separately for each thread. This prevents two or more threads from sharing this same control variable—a common-sense step that avoids threads stepping on each other.

Developers must make sure the body of the loop can be executed by different threads without collisions. One key requirement is that the threads not modify a common data item. Notice that in this loop, only one thread can ever access the elements of the respective arrays. For example, because OpenMP will break up the loop and consequently vary the starting and ending points, no two threads will ever touch the same elements.

Now if we look at the loop at lines 65-68 of the OpenMP version, we have a loop where this is not so. This loop reads through the array of shorts and for each short it encounters, it increments an element in an array of integers that count the frequency of that short. So, for example, if a short has the value of 2756, then element 2756 of values_array is incremented by 1. This gives us a count of how often each value occurred in the array of shorts. Using OpenMP, this loop would be broken up so that each thread had its own chunk of the array of shorts to process. However, despite this separation, it is possible that two threads would simultaneously encounter a short with the same value and would both try to update the corresponding counter at the same time. This would result in an error, due to the conflicting access. (To verify this, un-comment the OpenMP directive before this loop and rerun the program. You will see that the program results change.)

I should note that this problem is not unique to OpenMP. It occurs in all parallel programming. Anytime two threads might access the same variable, a conflict will occur that must be handled explicitly.

OpenMP and native methods enable us to enclose code with a mechanism that prevents the code from being executed by more than one thread at a time. However, because this approach forces the other threads to wait while execution completes, these steps destroy some of the benefits of parallelization. It is much better to design the loops and associated code to avoid synchronization conflicts. (Naturally, this cannot be done entirely, and eventually you will need threads to share a variable. OpenMP and native methods both provide the locking methods to enable this sharing.)

More Goodness

OpenMP provides far more than just loop parallelization, although this is its simplest construct to use. You can parallelize larger chunks and even exert very detailed control over threading operations—all without ever directly creating a thread manually.

If you want to exploit multiple cores and want a simple, portable way of doing so, I highly recommend you start with OpenMP. You are likely to find that this will be all you need for many of your applications. In some rare situations, of course, it's not the right tool for the job. I hope this article has helped you see the difference.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Coarse-Grained Vs. Fine-Grained Threading for Native Applications, Part 1

Learn when to choose coarse-grained threading for your application.

By Alan Zeichick

The question is simple, even if the answer are not: How do we, as software developers, keep up with Moore's Law? The speed of our hardware continues to increase, thanks to innovations like dual-core and multi-core chips, bigger and faster caches, improved microcode, enhanced buses, swifter memory, and so-on. However, it's not a given that application performance will keep pace with the faster hardware. This two-part article will explore these issues.

Let's say that you have two servers, and one has exactly twice the specWEB http://www.spec.org/benchmarks.html or TPC-C http://www.tpc.org/tpcc/default.asp benchmark performance of the other. Will your own n-tiered or client-server app run twice as fast on the second machine? Maybe… but maybe not. The same is true with desktop software, of course – unless you do some work to take advantage of the hardware, your apps may not scale to take advantage of the hardware.

One reason why apps may not scale the way you expect is because the traditional means for speeding up the computer – boosting clock speed – is hitting the wall. Improvements in clock speed, or in simple chip improvements like adding more cache, certainly does help performance, but only incrementally. It's like driving on the highway: Doubling the horsepower of your engine won't cut your drive time in half. (A great article that talks about this issue is "The Free Lunch is Over," http://www.gotw.ca/publications/concurrency-ddj.htm by Herb Sutter.)

Traditional application architecture is like your commute – linear, going from Point A to Point B. These single-threaded applications are easy to design, and easy to understand. They're easy to code, easy to test, easy to debug, easy to tune and easy to maintain. What's not to like? Well, unless you're running in a managed environment like Java or .NET, sequential applications won't take advantage of multiple processors in a server, or multiple cores in a processor.

By the way, I should point out that what I wrote above isn't strictly true – single-threaded linear applications *will* run faster on a multi-processor platform running a modern operating system, like Linux, Solaris or Windows. That's because the OS will schedule different parts of its own functions to all the hardware resources, and also allocate different applications to different processors and cores. This lets the single-threaded application run faster because it gets more cycles on the single processor core available to it. So, from that perspective, there's a performance benefit from running a linear, sequential application in a multi-core or multi-processor box. However, the application

itself isn't taking advantage of the hardware, and therefore performance gains will be rather minor.

A better approach is threading. When it comes to writing native code, the are two types of threading, one which I'll call coarse-grained, the other which I'll call fine-grained. Fine-grained threading is further broken up into two sub-types, explicit and implicit. Course-grained threading is generally always explicit. Let's talk about what I mean.

Course-Grained Threading

Most applications being developed today are either single-threaded or coarse-grained threaded. The coarse grain is where the threads are logical application functions, designed by the software architect. In a game, coarse-grained threads might watch input devices, like joysticks, or play the soundtrack, or generate special sound effects, or handle the display, or communicate over the network. For an e-mail client, coarse grains would handle fetching POP3 messages, sending SMTP messages, indexing the stored messages, searching that index, and rending plain-text or HTML messages in a window.

We're all familiar with these sorts of threads; they make applications responsive, and by architecturally breaking the app down into logical units, also make it easier to design, code, test, tune and deploy. Also, different development teams can potentially focus on just one of those coarse threads, helping you manage the software development life cycle more efficiently. (If each coarse-grained thread has its own memory space, it is sometimes referred to as a process.)

The reason that almost all of these coarse-grained applications are explicit is that an architect deliberately designs the threads and processes as part of the logical model of the application. There's little automation involved, other than perhaps what a code generator spits out, if you're going that route. Creating, managing, running and killing those threads is a manual, explicit process. You're also responsible for managing locks, and watching for race or deadlock conditions, but architecturally, if the number of threads is small, and if the logical separation in the functions is clear, this isn't difficult.

(A race condition is where one thread makes a change to a memory location that some other thread isn't expecting to change, and this changes the results. A deadlock is where one thread is waiting for another to release a lock – but the second thread is waiting for the first thread to release a lock. So, nothing happens.)

The good news, therefore, about coarse-grained threading is that it's easy to do. It's not rocket science. The bad news is that it's not hugely scalable. Each one of those coarse threads is, essentially, its own single-threaded process, and can't take advantage of scalability. For example, consider the e-mail client running on a multi-core box. Say there's a period of time when you're not checking for new e-mail or sending out

messages, and the HTML message on the screen has already been rendered. If the user starts a message search, the main routine spawns a search process – which is single threaded, and runs, let's say, on core #3. If the search takes 15 seconds, well, it takes 15 seconds, even if the other cores are fairly idle. Add another core… and it still takes 15 seconds.

Similarly, if the end user opens a message that contains a complex graphic to be rendered – maybe a large PDF is being previewed, and must be scaled and anti-aliased – that tasks is also probably running in a single thread, not taking advantage of all that the hardware can offer. So, even though other threads might be running at the same time (maybe new messages are being downloaded from a POP3 account), the rendering is only using a single core, and proceeds linearly.

Another bad news item is that coarse-grained applications typically don't scale upwards. How would that e-mail client take advantage of an eight-cored system? Or 16? Well, it wouldn't. Sure, it's silly to think about scaling an e-mail client across 16 cores – but not a game, or a scientific visualization application.

Part II will discuss fine-grained threading, and where it fits into the development process.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Coarse-Grained Vs. Fine-Grained Threading for Native Applications, Part 2

Learn when to choose fine-grained threading for your application.

By Alan Zeichick

In the first part of this two-part story, we discussed the need for threading applications. We discussed why single-threaded, or linear, applications don't scale to take advantage of new multicore processors and multi-processor computers. We also discussed that there are two different ways to scale.

Coarse-grained threading architecturally separates the application up into separate functional units, each of which runs in its own thread. (That thread might be referred to as a process, if it has its own memory space.) Separating applications into coarse-grained threads will certainly help a complex application, like an e-mail client, more efficiently do different things at the same time, such as retrieving messages from multiple POP3 mailboxes while also sending multiple SMTP messages while conducting a message archive search while rending an HTML e-mail message with an embedded graphic.

However, coarse-grained threading won't accelerate those individual tasks, providing them with the resources they need to handle complex tasks. So, if your message renderer has to scale a big PDF image to fit into a window, and if that message renderer runs in a single process, that task might take 15 seconds – whether you have 2 cores or 8 cores available. How do you make the individual tasks faster? Through either explicit or implicit fine-grained threading.

Fine-Grained Threading

Fine-grained threading is opportunistic parallelization within a logical processor task. Generally, it's not part of the application's architecture; functionally, it is (or should be!) invisible, with the only benefit being that the application code can "burst" into spawning short-lived threads to tackling a compute-intensive task, decompose the task into pieces that can run in individual threads on lots of processors to get the job done quickly, then merge those threads back together when the short-term task is done.

So, where coarse-grained threads would handle logical functions of the application like rendering messages or managing the message-search function, fine-grained threads would run within those bigger threads to implement a parallel search algorithms, or break up the scaling of a large graphic into a smaller one. In other words, they'd turn that 15-second search into a 5-second search or a 3-second search.

As mentioned before, there are two types of fine-grained threading, explicit and implicit. Both have their pros and cons – or as I like to phrase it, good news and bad news.

Explicit Fine-Grained Threading

Explicit fine-grained threading is hand-coded by the developer using a language like C/C++, with the help from threading libraries which can help with locks and memory management. The developer, however, would need to be able to identify those places in the code which could benefit from threading, and then make the calls to spin off the threads, each with their piece of work, while the main thread manages the code, allocates the memory, integrates the results, frees the resources and then kills the threads.

The good news is that in the hands of an expert developer – and explicit threading is best left to the experts with tremendous knowledge of the operating system, threading libraries and computer science – this technique can yield astounding performance improvements. That's why there are parts of critical applications, like games, or parallel math libraries, or graphics programs, that have key routines that have been hand-coded for optimal results. Such sections of code are identified using run-time profiling, or are tackled during refactoring exercises.

The bad news about explicit parallelization is that it does require an expert – and even then, it's difficult. Multi-threaded programs can be non-deterministic. It's very difficult to ensure that they're give the right results, or that they'll even terminate. This is where race conditions and deadlocks come into play; it's easy for one thread to stomp all over another thread here, and take down the application. Explicit fine-grained threaded routines are also typically coded (either intentionally or subconsciously) for a specific number of processor cores, and run-time results may suffer if the hardware has more or fewer codes than expected, or the operating system doesn't give the application the optimal resources.

Worse, most testing tools can't handle explicit fine-grained threading. Source code analyzers can't always detect problem conditions or tell you if the threads will yield the right results. Functional and stress tests may not exercise all the pathways, or detect subtle errors brought about by locking problems or thread sync errors. And don't get me started on memory leaks… explicit threading is a tool best used with caution. But again, it's your only route to the utmost in performance tuning. (We'll be covering explicit threading more on the AMD Developer Center in the future.)

Implicit Fine-Grained Threading

Implicit fine-grained threading is much, much simpler, and is the approach that I recommend for most developers. It uses a system like OpenMP http://www.openmp.org ,

which uses a series of in-code pragmas and code libraries to tell the compiler to automatically parallelize a piece of code.

OpenMP avoids almost all of the difficulties associated with explicit fine-grained threading, and is much easier to perform. OpenMP also available for every platform and every major compiler, thanks to its standing as a vendor-neutral interface for threading – and it's even free. On the other hand, OpenMP not suitable for all tasks, and doesn't offer as tight an optimization as hand-coded explicit threading performed by an expert.

To use OpenMP, you link in the appropriate library, and use code pragmas to indicate to the compiler areas where you think that the code can be parallelized. OpenMP excels at loops where there aren't external dependencies, and where the compiler can figure out how to optimize the code. It's not so good for situations where there aren't loops.

So, for example, you can use OpenMP to do things like breaking up the task of rendering and scaling a huge PDF file into a small window, if you have an algorithm that can break that task into smaller pieces. First, the main thread implements divides the graphic into chunks that can be rendered and scaled independently. Then, you write the code that does the rendering and coding, as if it were a loop that worked on each chunk in sequence – but use pragmas to tell OpenMP that this can be optimized. (You must take care to ensure that the code truly can be parallelized, and that there are no dependencies that can happen if the outer loop runs out of sequence.) After this code, the main thread integrates the results.

At compile time, the compiler will then automatically analyze the code and generate the correct code to spawn the fine-grained threads, let them work, and then integrate the results. If you've designed the algorithm correctly, you get the performance gain of explicit fine-grained threading, without the hard work, and with fewer opportunities for error. Best of all, the code remains easy to read and study. In fact, the compiler can generate both a fine-grained threaded and non-threaded version of the code, so you can stress test them to ensure that they're providing the same answers, and so you can use code profiles to compare the version to make sure it's working correctly.

Is OpenMP perfect? No. You still can code in race conditions or deadlocks, if the logic or algorithms are wrong. OpenMP is also only good for situations where the parallel tasks use the same code; think of it roughly analogous to loop unrolling on a massive scale. (You can thread other things beyond loops in OpenMP, but that's where it's usually applied.) We'll be writing more about OpenMP in the AMD Developer Center as well.

Tying Up the Threads

Here are four points to take away:

* Single threaded applications don't scale to take advantage of modern hardware and operating systems.

* Coarse-grained threads, which generally correspond to processes, help your applications perform better and do many things at the same time, but doesn't help them perform specific tasks faster using all hardware resources.

* Explicit fine-grained threading can offer tremendous performance advantages, especially in situations that require complex data or memory management, or where the application doesn't lend itself to implicit fine-grained threading using OpenMP. However, this work is difficult, requires considerable expertise, and in non-trivial implements can be extremely hard to test or debut.

* Implicit fine-grained threading, such as using OpenMP, makes it easy to improve application performance by getting rid of CPU-intensive bottlenecks. However, it's only suitable for cases where there are suitable algorithms, clear functional decomposition, and no chance of deadlock or other thread problems.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Optimizing for Multi-Core with AMD CodeAnalyst

Learn how to use CodeAnalyst to optimize application performance.

Justin Whitney

Optimizing for hardware is usually something that happens at a very low level. While AMD multi-core processors can dramatically increase the performance of any application, the biggest gains happen when developers can tune their code specifically for the processor, something high-level developers can have a hard time doing.

AMD CodeAnalyst changes that story. Visual Studio developers have a few cards of their own to play. Now CodeAnalyst helps them find the best place to play them. Using multiple profiling techniques, developers can find the bottlenecks in their code and measure performance from a variety of angles. And because CodeAnalyst actually plugs into Visual Studio itself, you can get real-time information while you code.

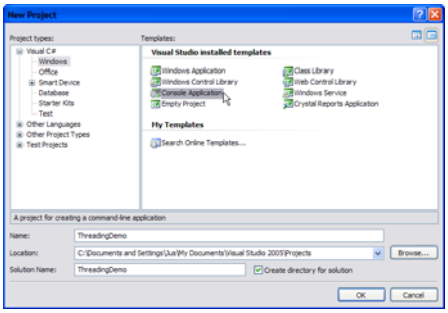

Figure 1. ThreadingDemo Console App

But enough exposition. How does this happen? What does it look like? To answer these questions and more, you're going to try a little experiment. Assuming you have Visual Studio 2005 installed, take a moment to download and install AMD CodeAnalyst [http://developer.amd.com/downloads.jsp]. Then you'll walk through an exploration of the tool itself, as well as a before and after analysis of a simple card shuffling app written in C#.

Shuffling Cards in C#

As a quick demonstration of CodeAnalyst, you're going to build a card shuffler. You won't have any graphics or game mechanics, just a few Deck objects and string arrays that get mixed around. This version in particular is designed to be a big time waster—how often do you need to shuffle a deck of cards 12,345,678 times, much less four decks in succession? Start Visual Studio 2005 and create a new C# Windows Console application. Call it ThreadingDemo (see Figure 1).

Ignore the Program class for now and add a new class to the project. Call it Deck. Replace the code for Deck with the code in Listing 1.

Listing 1.

Deck.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ThreadingDemo
{
    public class Deck
    {
        private string[] m_cards;
        private string m_deckOwner;

        public Deck(string deckOwner)
        {
            m_cards = new string[52];
            m_deckOwner = deckOwner;
            int thisCard;
            int cnt = 0;
            for (int suit = 0; suit < 4; suit++)
            {
                for (int num = 0; num < 13; num++)
                {
                    cnt = (suit * 13) + num;
                    switch (num)
                    {
                        case 0: m_cards[cnt] = "Ace";
                            break;
                        case 10: m_cards[cnt] = "Jack";
                            break;
                        case 11: m_cards[cnt] = "Queen";
                            break;
                        case 12: m_cards[cnt] = "King";
                            break;
                        default: thisCard = num + 1;
                            m_cards[cnt] = thisCard.ToString();
                            break;
                    }
                    switch (suit)
                    {
                        case 0: m_cards[cnt] += " of Spades";
                            break;
                        case 1: m_cards[cnt] += " of Hearts";
                            break;
                        case 2: m_cards[cnt] += " of Clubs";
                            break;
                        case 3: m_cards[cnt] += " of Diamonds";
                            break;
```


```
                    }
                }
            }
        }

        public void Shuffle()
        {
            int card1, card2;
            string tmpCard;
            Random RandomClass = new Random();
            for (int i = 0; i < 12345678; i++)
            {
                card1 = RandomClass.Next(0, 52);
                card2 = RandomClass.Next(0, 52);
                tmpCard = m_cards[card1];
                m_cards[card1] = m_cards[card2];
                m_cards[card2] = tmpCard;
            }
            Console.WriteLine(m_deckOwner + "'s hand...");
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine(m_deckOwner + " " +
i.ToString() +
": " + m_cards[i]);
            }
            Console.WriteLine();
        }

    }
}
```

Go back to Program.cs and replace its code with the code in Listing 2.

Listing 2.

Program.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ThreadingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Deck d1 = new Deck("Archie");
```

```
            Deck d2 = new Deck("Betty");
            Deck d3 = new Deck("Jughead");
            Deck d4 = new Deck("Veronica");

            d1.Shuffle();
            d2.Shuffle();
            d3.Shuffle();
            d4.Shuffle();
        }
    }
}
```

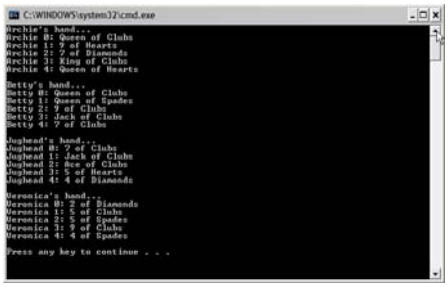

Figure 2. Results of Non-Threaded Shuffle

Before you hit Go, there are a couple of details worth noting. First off, you can probably think of half a dozen other ways to build a deck of cards and shuffle them. But this technique in particular was chosen to showcase the use of the Threading class, which you'll see in a moment. A good rule of thumb: when you have the chance to design apps from the ground up, use that opportunity to code them in a way that takes advantage of multi-core performance gains by building optimization techniques into your architecture at the earliest possible stage.

Second, you might notice that in the Shuffle method of the Deck class, cards are being shuffled an awful lot. If you're running on an older processor, feel free to reduce that number to something more manageable. This demo is being done on an AMD Athlon 64 X2 Dual Core and requires an unreasonably huge number of loop iterations for the human eye to even register that something happened.

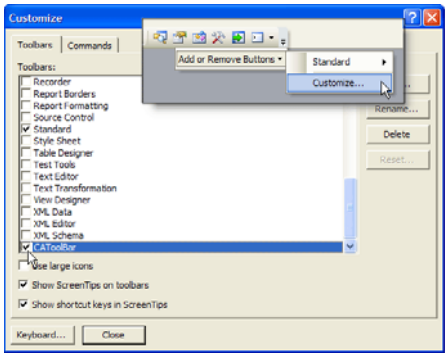

Figure 3. Adding CodeAnalyst Buttons

Build Without Debugging (Ctrl-F5) so you can see the results before the console closes. Depending on the speed of your processor, it should run for a few seconds and look something like Figure 2.

Your First CodeAnalyst Project

The AMD CodeAnalyst Performance Analyzer [http://developer.amd.com/downloads.jsp], which comes in both Windows and Linux flavors, uses a suite of profiling tools to give you different perspectives on your app's performance. The Windows version [http://developer.amd.com/cawin.jsp], covered here, integrates with Visual Studio 2005. In fact, when you first install CodeAnalyst, it looks for a version of Visual Studio 2005 and automatically installs an Add-in for it. When you load Visual Studio, you may not see it at first. To find the CodeAnalyst toolbar, either right-click on the main toolbar area or click the dropdown to "Add or Remove buttons," then customize, as shown in Figure 3. In the list of toolbars, find and check CAToolbar, probably at the bottom of your list.

Figure 4. CAToolbar Ready to Begin

When the CodeAnalyst toolbar loaded, you can perform many of the same functions as when you run CodeAnalyst independently. When you change settings and specify a project location, you'll see the Start Profile button enable, which allows you to perform profiling from within VS. Figure 4 shows you what the toolbar looks like once it's ready to run.

The rest of this walkthrough will run from CodeAnalyst itself. Install CodeAnalyst [http://developer.amd.com/downloads.jsp] if you haven't already and create a new project.

- Session Name: you can leave the default. This is the prefix for the various profile sessions that you'll be running, so you can change it to indicate a particular profile, for example.
- Project Directory: don't be confused—this is the directory where all the CodeAnalyst data files will be saved, NOT the directory for your VS project. Navigate to a fresh directory where you want your sessions saved.
- Project Name: can be anything. Call this one ThreadingDemo.

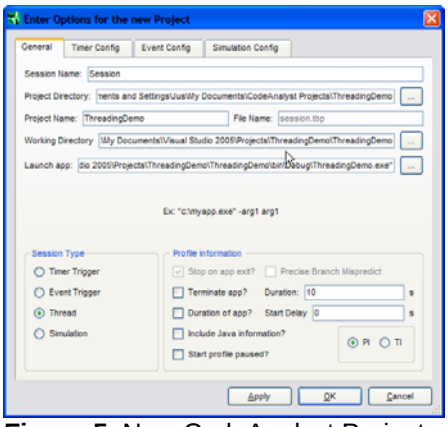

Figure 5. New CodeAnalyst Project

- Working Directory: This is the directory where your VS project files can be found. Navigate to the project folder that holds the \bin\ folder and .cs files for your project.
- Launch app: the actual executable you want to test. Navigate to \bin\Debug\ThreadingDemo.exe.
- Session Type: here's where you can select one of four different profiles. Select Thread. See below for descriptions and screenshots of the various types.
- Profile information: when you selected a Thread session type, then "Stop on app exit" should've become checked. So leave all the defaults here.

Your new project should look something like Figure 5. Note: here's a quirk to watch out for. If you hit Apply, then CodeAnalyst creates the new project subdirectory where you specified under "Project Directory." However, if you then hit Ok, it tries to do it again and gives you an error message that the directory already exists. Just ignore the error.

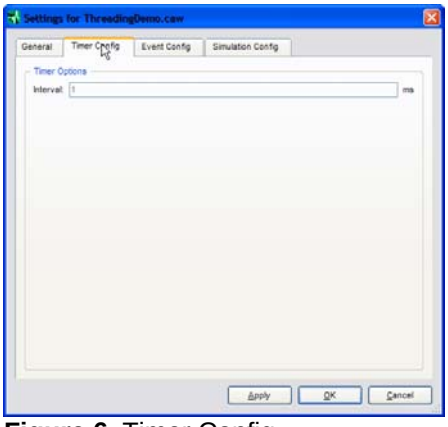

Figure 6. Timer Config

CodeAnalyst Session Types
CodeAnalyst gives you four types of profiling sessions. You can set the session type for a project at its creation or later, by selecting Tools -> Project Options. Session data from previously selected types will be kept in the project, giving you the chance to compare several types of profiles at once.

Timer Trigger Session
The Timer Trigger is used to capture Timer-Based Profiling (TBP) data, specifically from multiple processors in a multi-core system. When you run your app, CodeAnalyst collects samples at predetermined intervals, which default to 1ms and can go as low as .1ms. This interval can be changed in Project Options under the Timer Config tab, as shown in Figure 6.

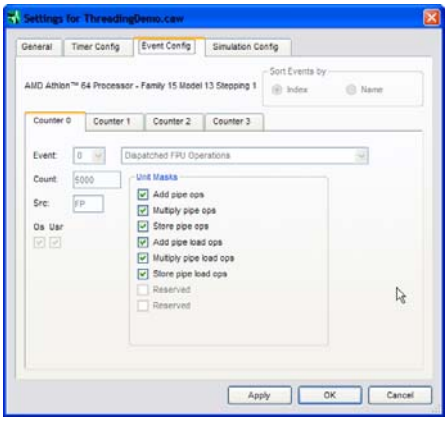

Figure 7. Event Config

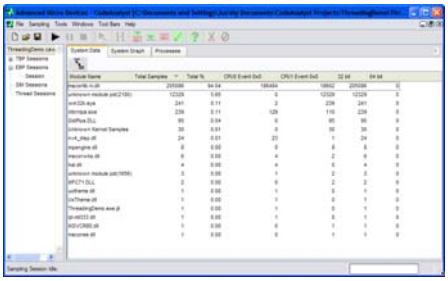
Figure 8. Event Session

Event Trigger Session
Selecting the Event Trigger will allow you to capture profiling based on up to four different hardware events. Configure these events through the Event Config tab of the project properties, as shown in Figure 7. This shows you potential bottlenecks, as in Figure 8, giving you a starting point for more detailed code optimization.

Pipeline Simulation Session
As seen in the Simulation Config in the project properties (Figure 9), you can run pipeline simulations for the AMD Athlon, AMD Athlon XP, AMD Opteron, and AMD Athlon 64. After selecting the target processor, set trace points in the source code, which comes up in CodeAnalyst UI. After execution is traced, it is simulated on the chosen processor. Note that this works only for single processor systems on unmanaged code. It doesn't apply to .NET code such as you see here.

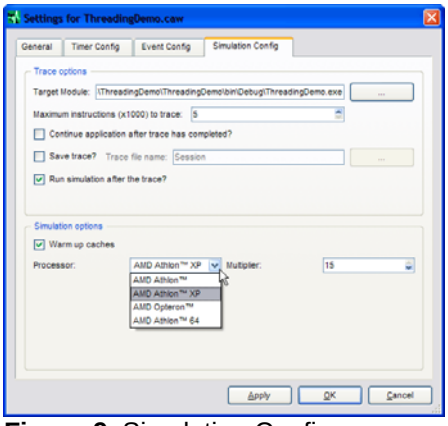

Figure 9. Simulation Config

Thread Session

The Thread Profile gives you both thread access charts and non-local memory access profiles. In other words, it shows you how many threads are running and how well they're running in parallel. In the case of Dual-Core processors, this includes side-by-side CPU profile for each thread. In fact, this type of session will be the subject of the remainder of the walkthrough.

Reviewing a Thread Profile

If you're following along, then you've already set the session type for this project to Thread. Run the profile by choosing Sampling -> Start or clicking the Start icon on the menu bar. The console window should appear briefly and show you similar results to your earlier test. Only in this case, the app will exit on its own. If CodeAnalyst is exiting the app before it finishes, go back to the Project Options, check "Terminate App?" and increase the duration.

To be sure you're getting consistent data, run the profile two or three times. On the system used for this demo, the Thread profile looks like Figure 10.

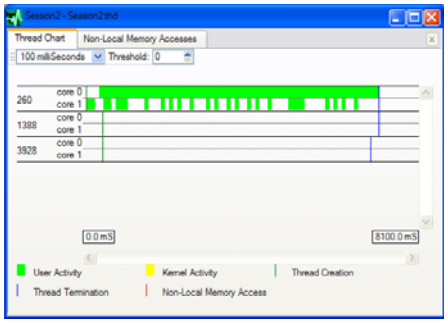

Figure 10. Non-Threaded Results

You should notice several important details here:

- In this example, the entire process is running on what is essentially a single thread. On other occasions, depending on the environment at the time of execution, more threads may be used, based on how the processor distributes the load.

- The length of execution time can change based on several factors, such as additional apps running in the background and even, in the case of laptops, whether or not the laptop is plugged in, due to system configuration related to power efficiency. (As an experiment, this demo was run on a laptop once using battery power, then again while it was plugged in. Execution time went from 14.7 seconds to 7.3 seconds—cut cleanly in half solely based on power management.)
- Although Visual Studio developers can't directly manage execution across multiple cores, a dual-core chip will automatically balance the load somewhat. But as you can see here, you still have a big opportunity to help the process along through creative code optimization.

Shuffling on a Thread

Now that you have the before picture, it's time to make a few changes and see what happens. First, close your CodeAnalyst project to free up any locks on the \obj\ directory. Then go back to your ThreadingDemo solution.

Replace the source code for Program.cs with the code in Listing 3. This includes a new reference to System.Threading, as well as use of the ThreadStart Delegate [http://msdn2.microsoft.com/en-us/library/system.threading.threadstart.aspx] and the Thread Class [http://msdn2.microsoft.com/en-us/library/system.threading.thread.aspx].

Listing 3.

Program.cs with Threading

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace ThreadingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Deck d1 = new Deck("Archie");
            Deck d2 = new Deck("Betty");
            Deck d3 = new Deck("Jughead");
            Deck d4 = new Deck("Veronica");

            Thread shuffle1 = new Thread(new
ThreadStart(d1.Shuffle));
            Thread shuffle2 = new Thread(new
ThreadStart(d2.Shuffle));
            Thread shuffle3 = new Thread(new
ThreadStart(d3.Shuffle));
            Thread shuffle4 = new Thread(new
ThreadStart(d4.Shuffle));
```

```
            shuffle1.Start();
            shuffle2.Start();
            shuffle3.Start();
            shuffle4.Start();
        }
    }
}
```

Hit Ctrl-F5 to make it go. In Figure 11 you'll see the results on the AMD Dual-Core processor.

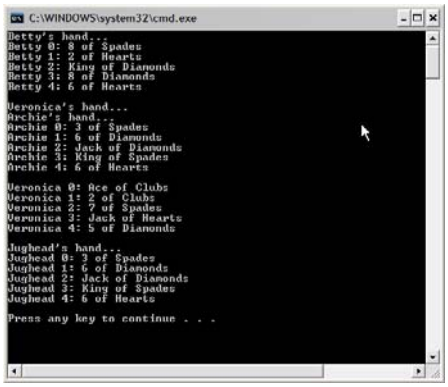

Figure 11. Results of Threaded Shuffle

Some really interesting (and exciting) things to note here:

- Running multiple independent threads can result in unpredictable behavior. In this case, since each thread wrote to the console on its own, the result ended up a bit jumbled—compare this to Figure 2 and notice how Veronica and Archie are mixing it up. Your results may be even more jumbled.
- More significantly, two of the threads kicked off so fast that they ended up using an identical randomization seed, which is based on the system clock by default. As a result, Archie and Jughead have the exact same hands. Just something else to keep in mind, especially if you're doing any randomization.

Analyzing the Threaded Shuffle
Go back to CodeAnalyst and open up the project. You don't need to change a thing—just kick it off as-is. (Savvy readers may have noticed that the previous test run also serves to load any necessary .NET Framework business, as generally happens on the first execution of an app.)

As before, execute the app three times. You'll notice that the threads are distributed differently each time but with the same basic results, seen in Figure 12.

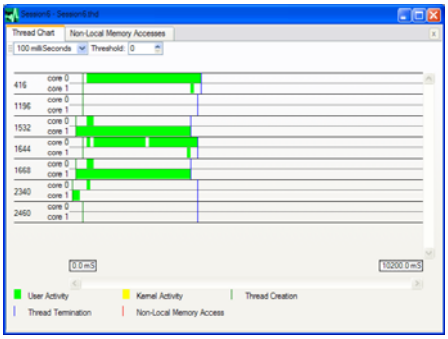

Figure 12. Results of Threading Optimization

As you can see, the results are pretty startling:

- Except for a few extra threads for overhead, the app is cleanly distributed among the four threads specified in the updated code.
- In addition, the code optimization very clearly takes advantage of the AMD Dual-Core processor. Look closely at the chart—threads 416 and 1644 run on Core 0 while threads 1532 and 1668 are running simultaneously on Core 1.
- This app is now going stupid fast. It went from 7.3 seconds to 3.8 seconds, again cutting execution time in half (a performance boost of 47.9% to be exact).

Where to Go From Here

In the right hands (yours), CodeAnalyst can be a boon to apps being optimized for a multi-core architecture. This walkthrough barely begins to show you the kind of optimization techniques available. As you begin exploring the tool, check out these resources for walkthroughs on the other profiling techniques, as well as additional coding tips for AMD Dual-Core processors.

- AMD CodeAnalyst Performance Analyzer for Windows
- An Introduction to Analysis and Optimization with AMD CodeAnalyst [http://developer.amd.com/articles.aspx?id=2&num=1]
- Got Bottlenecks? CodeAnalyst Can Help
- Managed Threading Basics [http://msdn2.microsoft.com/en-us/library/hyz69czz(VS.80).aspx]
- Multi-threading Best Practices [http://msdn2.microsoft.com/en-US/library/1c9txz50.aspx]
- Taking Game Performance to the Max with Threading [http://developer.amd.com/articles.aspx?id=4&num=1]

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Taking Advantage of Concurrent Programming for Windows, Part 1:

Learn the simplest multi-core parallelism that works.

Larry O'Brien

Many programmers don't understand this simple reality: When using mainstream programming languages the only way to take advantage of multiple cores is to explicitly use multithreading. Mainstream language processes are not automatically parallelizable, and there is very little that the compiler is *allowed* to do to exploit this new and exciting era of concurrent processing.

There are of course speedups from the operating system and some libraries can be rewritten to fulfill their contracts asynchronously, but *your* code is not going to run faster unless you make it multithread. The "free lunch" performance boost of increased clock speeds of previous processor generations is over, and while single-threaded applications will certainly experience some incremental improvements, performance-oriented programming will increasingly rely on distributing the calculations over multiple cores and processors using multiple threads.

Resources

- Download the Code for this Article

Other articles in this series:

- Taking Advantage of Concurrent Programming for Windows, Part 2: Multi-Core Programming in .NET
- Taking Advantage of Concurrent Programming for Windows, Part 3: Locality Effects in .NET Multi-Core Programming

But multithreading is hard, right? Race conditions, deadlocks, scheduling algorithms—any discussion of multithreading has to emphasize the insidious assumptions we make, how intermittent and difficult it is to isolate the defects, how difficult the debugging, and so on. Yes, it's all true, but let's shove that out of the way and talk about situations where multithreading is easy.

It's perhaps surprising that C++, with its reputation for difficulty, actually provides one of the easiest ways to exploit multi-core and multiprocessor systems. OpenMP, a multiplatform API for C++ and Fortran, uses compiler instructions to automatically generate all of the support code needed to parallelize code sections. In the simplest case, which is what we're going to focus on for this article, simply wrapping a processor-intensive loop in a #pragma block can lead to about a 70 percent performance increase on a dual-core or dual-processor system and enjoy a similar "free lunch" on the quad-core systems that you build in the future.

Figure 1. The Transform Steps

You can exploit concurrency at any level of granularity from high-altitude system architecture to CPU registers, but the key to avoiding trouble is always the same: minimize data coupling. To err is human, but to really screw up you need shared state. This is why it's often easiest to accomplish concurrency at the highest level (n-tier and service-oriented architectures) where big chunks of functionality can be divided up and coordinated with coarse messages. Meanwhile, successful threading of middle-level abstractions (active functions or classes) is often fraught with difficulty.

The same principle leads to the seeming paradox that the other often-easy win for concurrency is at the lowest level with a loop working on an array. In the center of many, perhaps even most, processor-intensive scenarios, lays a hotspot involving a loop and a big array of data: the pixels, the sound bits, the mesh coordinates, or whatever monstrous array of data you may need for your work. And while there's certainly a chance of encountering a concurrent pitfall in such a situation, but there are also many cases in which the calculations are nicely independent.

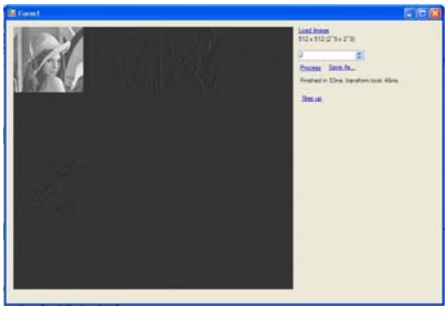

Figure 2. Wavelet-based Image Processing

Image processing is typical of the work you might tackle with OpenMP. Wavelets are fascinating tools for image processing because they can be used for compression, feature detection, enhancement, and probably dozens of other techniques. The simplest wavelet, the Haar wavelet,

works by calculating the average and difference-from-average of pairs of input data. These transformations are then gathered together, reordering the data. **Animation1** shows the Haar transform in action on a 1-dimensional array. On an image, we can perform one transform horizontally and then another transform vertically. When the difference coefficients are scaled into the grayscale range 0..255, the transform steps look like those shown in Figure 1. The result is that the original image is divided into four quadrants; the top-left is a half-resolution version of the original, and the others are essentially edge-maps. If the wavelet is applied recursively, the data quickly becomes un-interpretable to the eye, but details from multiple resolutions are captured in the transformed data. It's also important that the transform is lossless: the original image can be reconstituted perfectly by applying the transform in reverse.

I wrote a simple C++/CLI program to experiment with wavelet-based image processing (Figure 2). While it works quickly enough with the famous 512x512 Lena compression benchmark shown, on a 16megapixel panorama it can take a few seconds. Figure 3 shows CPU utilization during such a challenging run. This type of disappointing perfmon profile is more common than not because so few current applications are written for multi-core machines.

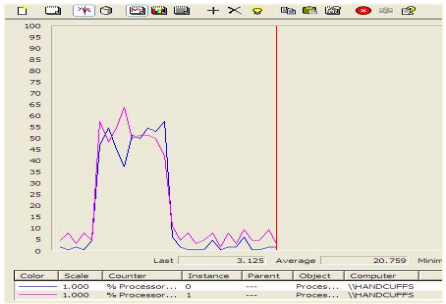

Figure 3. Before—CPU Utilization During a Challenging Run

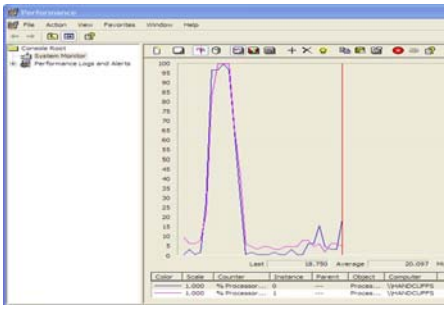

Figure 4. After—A More Pleasing Profile

I added two lines of code (one to each of the horizontal and vertical transform functions) and generated the far more pleasing profile shown in Figure 4. Both processors at near 100 percent utilization—resulting in a 70 percent speedup: now *that's* the sort of performance improvement you'd expect from multi-core! The magic is in those two lines of code.

The horizontal transform is shown below in Listing 1. As you can see, the listing follows the animation in structure. We iterate over the pOriginal array, calculate the average and difference-from-average, and store the results in the appropriate position in the pNew array. (Complete source code and binaries are available for download here.

Listing 1.

```
void HorizontalStepDown()
{
#pragma omp parallel for
       for(int y = 0; y < height; y++)
       {
              int yOffset = y * width;
              //Note x is being stepped 2 at a time
              for(int x = 0; x < width; x+=2)
              {
                     float v0 = pOriginal[yOffset + x];
                     float v1 = pOriginal[yOffset + x + 1];
                     float ave = (v0 + v1) / 2;
                     float diff = v0 - ave;
                     pNew[yOffset + x / 2] = ave;
                     pNew[yOffset + x / 2 + width / 2] = diff;
              }
       }
}
```

The magic sauce is the #pragma line. Although there are a number of OpenMP utility functions, the majority of OpenMP use is done in the form of #pragma commands. These commands, which are ignored by compilers that do not support OpenMP, dramatically change the semantics of the code block to which they are applied.

Loops cannot, in general, be run in parallel. Many loops have loop-carried dependencies, such that correctness requires sequential processing. For example, if your loop contains lines like x[i] = x[i - 1]; you've likely got a loop-carried dependency. Naturally, the language must solve the general case and leave parallelization to either sophisticated compilers or to (presumably equally sophisticated) human beings.

An OpenMP parallel block is one that the programmer asserts is safe to run concurrently. As mentioned earlier, this is often easiest at the lowest level of abstraction, when you know that your task is to work your way through some big block of data. The OpenMP pragmas define the blocks that can be run concurrently, but they do have overhead. When execution first encounters a parallel region, some amount of threads are started (they were created on program start-up). Parallel regions are generated as out-of-line functions so that their addresses can be passed to the executing threads. So, on a single-core machine, OpenMP introduces overhead and gains nothing.

Even on a multi-core machine, it takes many iterations over a loop before the benefits of distributing the computation overtake the overhead of setting up the threads. In practice, any kind of media processing is likely to involve enough data to make opening threads well worthwhile. Even on the "Lena" image, which is a mere quarter-megapixel file, OpenMP delivered nearly 70 percent better performance. There will always be some overhead to distributing and coordinating concurrent operations, so multiple cores will never lead to perfect performance multiples.

In Visual C++ 2005, using OpenMP is as simple as adding the #pragma and compiling with a command-line switch (/openmp). In Visual Studio 2005, this can be set in the Project Property

Pages dialog, under "Configuration Properties|C/C++|Language|OpenMP Support" (see Figure 5). At the moment, Microsoft's OpenMP implementation is not compatible with the /clr:pure or /clr:safe command-line switches, so if you're writing in C++/CLI, you'll have to use the "plain vanilla" /clr switch which can also be set in the Property Page dialog, as shown in Figure 6. The OpenMP runtime file vcomp.dll must be shipped with the final executable code.

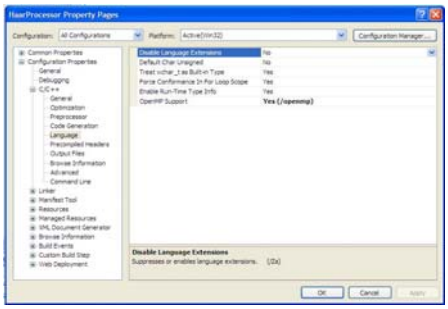

Figure 5. Enabling OpenMP Is an Easy Configuration Change

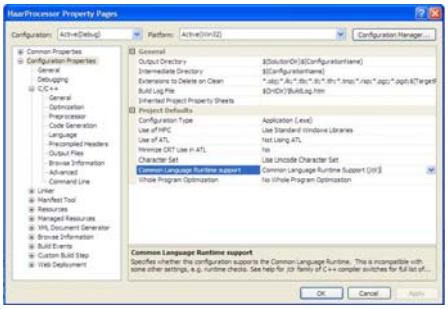

Figure 6. OpenMP Is Not Yet Compatible with /clr:Pure or /clr:Safe

Using OpenMP with C++/CLI is a joy, since you can use .NET's Base Class Library to make such things as GUIs, databases, and Web Services easy, but still be a mere OpenMP pragma away from unleashing your processors. In writing the sample application for this article, though, I noticed that the BCL's Bitmap class had abysmally slow SetPixel() and GetPixel() operations. Listing 2 shows how easy it is in C++/CLI to combine an easy-to-use BCL class like Bitmap with some low-level pointer work to speed things up. The only thing that's a little tricky is that the BitmapData returned by Bitmap->LockBits() should be released as soon as practical and, to be safe, should be wrapped in a try…finally block. A similar function for setting pixels is in the sample application source.

Listing 2.

```
void ImageToFloatArray()
{
        //getPixel is absurdly slow, so let's do it fast
        GraphicsUnit guPixel = GraphicsUnit::Pixel;
        RectangleF^ boundsF = myBmp->GetBounds(guPixel);
        Rectangle bounds = Rectangle((int) boundsF->X, (int) boundsF->Y,
(int) boundsF->Width, (int) boundsF->Height);
        BitmapData^ bmpData = nullptr;
        try{
```

```
                bmpData = myBmp->LockBits(bounds,
ImageLockMode::ReadWrite,
PixelFormat::Format8bppIndexed);
                unsigned char* pData = reinterpret_cast<unsigned
char*>(bmpData->Scan0.ToPointer());

                pOriginal = new float[width * height];
                pNew = new float[width * height];
#pragma omp parallel for
                for(int y = 0; y < height; y++)
                {
                        for(int x = 0; x < width; x++)
                        {
                                unsigned char pixelR = pData[(y * width +
x) *
sizeof(unsigned char)];
                                float grayScale = (float) (pixelR /
255.0);
                                pOriginal[y * width + x] = grayScale;
                        }
                }
        }finally{
                if(bmpData != nullptr){
                        myBmp->UnlockBits(bmpData);
                }
        }
}
```

It's important to realize that OpenMP is not a substitute for between-the-ears optimization. For instance, the code presented in this sample application uses floating point numbers, but since the Haar wavelet is limited to division by two, it's an easy-enough matter to implement the transform using integers and shifts. Sure enough, such a change doubles the speed of the transform, outshining OpenMP. The point of the sample application isn't to show pedal-to-the-metal optimization, but rather the opposite—how easy it is to get a significant performance boost with minimal work. Of course you can use an OpenMP pragma on the integer version of the transform and get a free lunch on that, too.

Right now, when the question is whether you have one core or two, the boost you can hope to get from an OpenMP pragma hovers around 70 percent. However, once we get four cores on our desktops, that same line might give you a 300 percent speedup. Wait a generation, and you might be talking about OpenMP delivering six times the speed of a single-threaded application. Beyond that, what about when machines start having 16 and 32 cores? Today you *might* be able to get by without parallelizing your code, but that's certainly not going to be the case in the not-so-distant future.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Taking Advantage of Concurrent Programming for Windows, Part 2

Learn coding techniques for multiple cores.

Larry O'Brien

Most programmers are wise enough to know that software won't automatically run significantly faster just because the user has a dual-core processor. While there will certainly be incremental benefits of the newer processor, the expected benefits of, say, doubling power based on adding a second core, can only be achieved with explicit code changes. No mainstream language can automatically take advantage of multiple cores and multiple processors. This is true of all the .NET languages from Microsoft except for F#. However, it's also true of Java, C++, C, Delphi, Ruby, JavaScript, Perl, and Python, just to name a few.

Should you care? Well, today, the question of changing your programming to exploit multiple cores is a question of whether your target machine has one or two cores available. The boost that you can hope to get from explicit multithreading is something less than 100 percent. Perhaps, for some, that gain is not worth the effort. Tomorrow, it will be common for there to be one, two, and four core targets. In just a few Moore's generations, the possibilities will stretch from one core to eight, 16, and even 32 cores. Today, we're in a transitional "multi-core" era, but in the subsequent "many-core" era it will be impossible for professional programmers to hide their heads in the sand when it comes to concurrency.

The Common Language Runtime (CLR) of .NET provides a fairly easy-to-use object-oriented threading model. A **Thread** encapsulates the idea of a "thread of execution." At the .NET level of abstraction, one can say that at any given moment, a single **Thread** is executing on a single processor core (once you get under the level of the OS and "closer to the metal," the problem of "what's going on" at any given moment becomes more complex). **Thread**s run within an **ApplicationDomain**, which is .NET's "unit of isolation for an application." To quickly finish the model, Windows (like most Operating Systems) has the idea of a "Process," each of which has its own memory space, security token, holds handles to resources likes files and windows, and contains one or more threads. Windows "multitasks" by allocating CPU time to different Processes and switching context between their threads them (and, because Windows can interrupt a Process without that Process's cooperation, it is said to be "preemptive"). In addition, Windows natively has the ability to coordinate threads, so it is a "preemptive multitasking, multithreaded" OS. An **ApplicationDomain** is, in reality, a lighter-weight entity than a Windows Process, but like a Process, has the important detail of sharing memory between its component **Thread**s.

I like to illustrate the benefits of multithreading on a multiprocessor machine using an image-processing example. Wavelets are a fascinating topic in signal processing: they can be used to perform compression, image enhancement, feature detection, and probably a dozen other things. The Haar wavelet is the simplest wavelet and works by calculating the pair-wise average and difference-from-average of the original signal (sound, image, video, etc.). For instance, if the original data was [8,6], the Haar wavelet would transform it into [7, 1]. [8,6,5,9] would be transformed into [7, 7, 1, -2] (as shown in **Animation 1**). Typically, the wavelet is applied recursively until the first value in the signal is equal to the average of the entire signal (so [8,6,5,9] becomes [7, -0.5, 0, 1.5] after 2 levels).

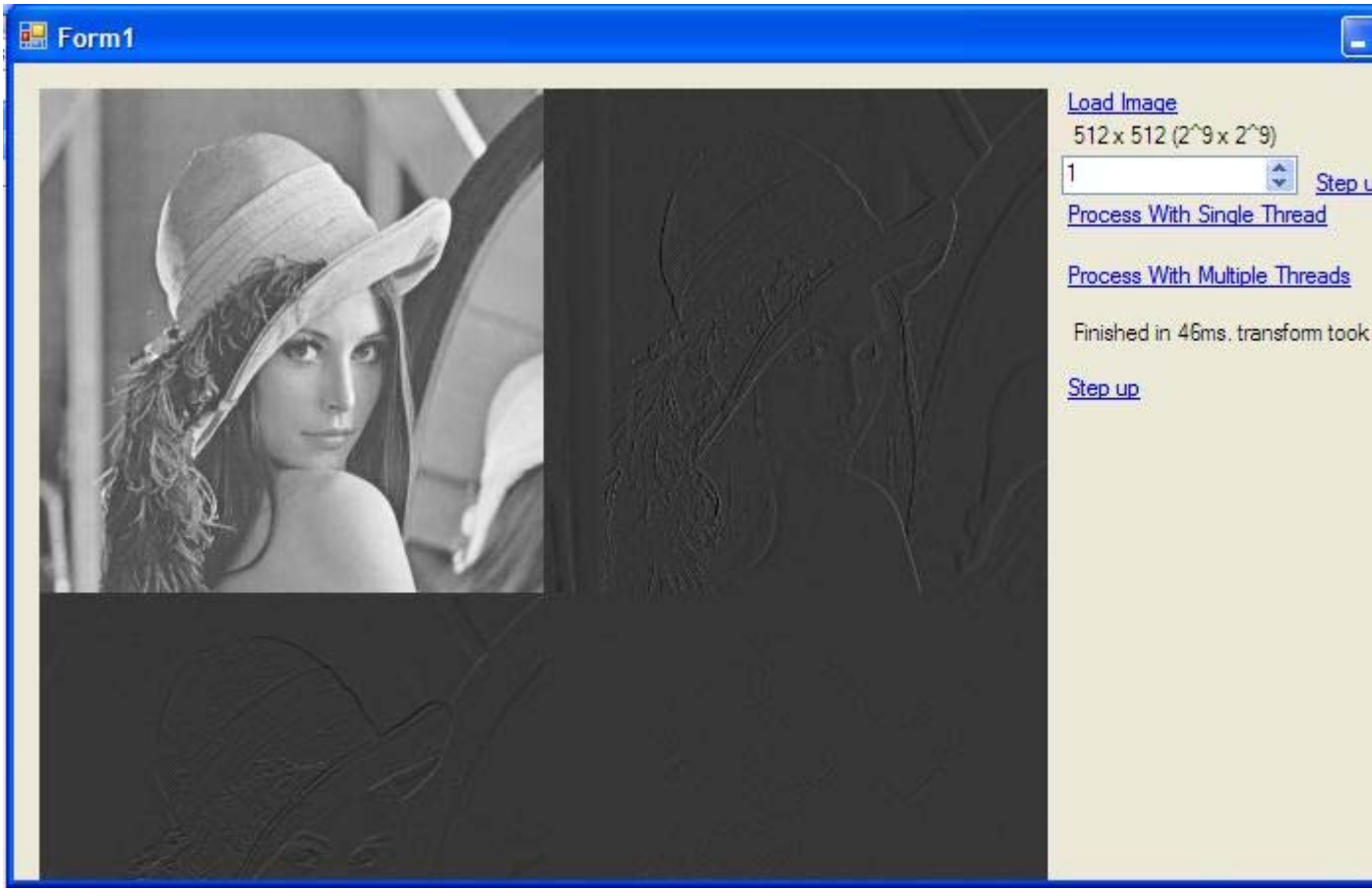

Figure 1. A C# Application That Applies the Haar Wavelet to an Image

Figure 1 shows a C# application that applies the Haar wavelet to an image. In this case, the image is the "Lena" compression benchmark image and the Haar wavelet has been applied once in both the horizontal and vertical dimensions, creating a half-resolution version of the original and 1-pixel edge maps. As you can see, applying a single level of transform to a 512 x 512 image takes only a dozen milliseconds or so, but Figure 2 shows that on a large panorama (4096 x 4096), the single-threaded implementation takes almost 8 seconds to complete a complete transform. Worse, Perfmon shows that during those 8 seconds I was hardly using one of my processors at all!

Listing 1 shows the C# code that performs a single Haar "step" in the horizontal direction. (Essentially, this is the same code as in Animation 1.) Listing 2 shows the same code transformed for multithreading. It's complex, but don't worry, we're going to provide an easier-to-use solution in a bit. In Listing 2's **HorizontalStepDownMT()**, we're going to use a **ManualResetEvent** to signal when the entire picture has been finished; we'll do that when the value of **remainingRowsPlusOne** becomes 0 and, at the beginning of the calculation, we set that value to 1.

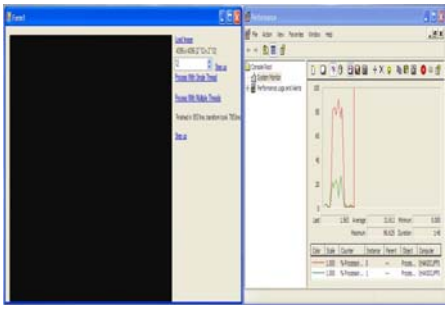

Figure 2. The Single-Threaded Implementation of a Large Panorama Has Lousy Performance

Then, we define the **horizontalTransform** instance of .NET's **WaitCallback** type. This delegate receives as a parameter the offsets to the pixels it should process and then performs the Haar transform (just as in the inner-loop in Listing 1). After the calculation is performed, the value of **remainingRowsPlusOne** is decremented in a thread-safe manner using **Interlocked.Decrement()**. The use of **remainingItemsPlusOne** inside the delegate is an example of what's called a "closure" or "outer variable capture."

Remember, at this point, all we've done is *defined* the **horizontalTransform()** function, we haven't *called* it. So now, we iterate over **y** calculating the offset corresponding to the first pixel in row **y**, just like the outer loop in a href="javascript:showSupportItem('listing1');">Listing 1. As we loop, we increment **remainingRowsPlusOne**. Then, instead of performing the calculation by directly calling **horizontalTransform()**, we pass that delegate and the **offset** as parameters to the .NET function **ThreadPool.QueueUserWorkItem()**. In production code, the **bool** result of **QueueUserWorkItem()** should be checked for success and, on failure, either retried or cause an exception to be raised.

Behind the scenes, this will cause the CLR to manage execution of **horizontalTransform()** delegates with the appropriate **offset** on a Windows thread. The call to **QueueUserWorkItem()** returns asynchronously, which is to say that it doesn't wait for **horizontalTransform()** to actually execute (I get bugged when people say that asynchronous calls "return instantly," but they do return pretty darn quickly). Another strategy would be to explicitly create the **Thread**s and handle their lifecycles manually, which isn't very hard, but the **ThreadPool** class is convenient and handles thread creation and destruction for you, based on machine resources.

After the loop is completed, we decrement **remainingRowsPlusOne** and, if its value is 0, we call **set()** on our **ManualResetEvent**. The final line in the function calls the **ManualResetEvent.WaitOne()** function, which does not return until **set()** has been called on the **ManualResetEvent**.

Do you see why **remainingRowsPlusOne** has to be set to 1 initially and the final **Interlocked.Decrement()** call is needed outside of the loop on **y**? Remember that we're queuing up instances of the **horizontalTransform()** delegate to be executed as soon as possible on whatever processor is available. It's possible (not likely, but definitely possible) that the very first execution of the **horizontalTransform()** will reach the call to **Interlocked.Decrement()** before the **y**-loop reaches the *second* call to **Interlocked.Increment()**. If that happened, the **ManualResetEvent** would be **set()** at that time. The **y**-loop would continue and all the

horizontalTransform() delegates would be nicely queued up, but the **WaitOne()** call wouldn't cause execution to block. Therefore, the **HorizontalStepDownMT()** function would return, even if there were still calculations in the queue. Any functions that assumed the calculations were finished would be in a "race condition": behavior would depend on the order in which the two threads raced forward. That's a bad thing, since the outcome of the race could very well vary from machine-to-machine and run-to-run.

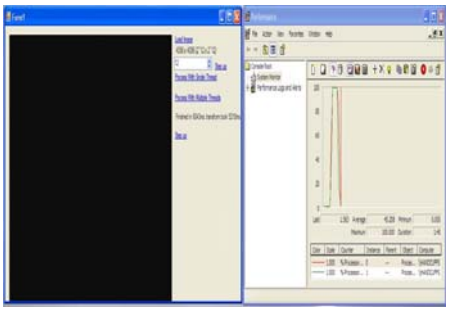

Figure 3. View the Difference Between the Single-Threaded and Multi-Threaded Version of This Program: 5.2 Seconds vs. 8 Seconds

Figure 3 shows the difference between the single-threaded and multi-threaded version of this program: 5.2 seconds versus 8 seconds (a 54 percent speedup). More importantly, Perfmon shows that both CPUs were pegged to the wall, so if I had quad-cores or more, my performance would continue to increase dramatically. (Download the source code for this application.) It should be noted that the code has functions that are dramatic improvements on the standard .NET **GetPixel()** and **SetPixel()** functions; these speedups use pointers and therefore the code needs to be compiled with the "/unsafe" compiler flag. The multithreading aspects of the code are CLR safe.) Further improvements could be made in a number of ways: queuing up larger chunks of work such as half a dozen lines per work item or transforming the algorithm to use integers (since it relies only on dividing by two and addition).

Virtually all of the discussion of Listing 2 was about threading infrastructure: very little was about the actual Wavelet transform. Listing 3 shows a more generic function called **ParallelApply()** that uses the **ThreadPool** to queue up execution of a **WaitCallback()** delegate on any **Ienumerable** collection. (Thanks go to Barry Kelly for helping improve the efficiency of my initial implementation.)

Any discussion of parallel programming has to acknowledge the difficulties: changing your mental model of how code is executed; race conditions; deadlocks; debugging, etc. These are all true. But often these warnings overshadow the often straightforward and sometimes downright easy scenarios. Since so much performance-oriented programming involves applying a calculation to large amounts of data that isn't interdependent (or whose dependencies can be factored out, as we did here), multithreading can often be done fairly easily.

Larry O'Brien is a recognized expert on .NET, and is a frequent writer and speaker on software development.

Listing 1.

```
void HorizontalStepDown()
{
       for(int y = 0; y < height; y++)
       {
              int yOffset = y * width;
              //Note x is being stepped 2 at a time
              for(int x = 0; x < width; x+=2)
              {
                     float v0 = original[yOffset + x];
                     float v1 = original[yOffset + x + 1];
                     float ave = (v0 + v1) / 2;
                     float diff = v0 - ave;
                     newValues[yOffset + x / 2] = ave;
                     newValues[yOffset + x / 2 + width / 2] =
diff;
              }
       }
}</
```

Listing 2.

```
void HorizontalStepDownMT()
{
    ManualResetEvent done = new ManualResetEvent(false);
    int remainingRowsPlusOne = 1; //Yes, 1.

    WaitCallback horizontalTransform = delegate(object state){
        int offset = (int)state;
        //Note x is being stepped 2 at a time
        for (int x = 0; x < width; x += 2)
        {
            float v0 = original[offset + x];
            float v1 = original[offset + x + 1];
            float ave = (v0 + v1) / 2;
            float diff = v0 - ave;
            newValues[offset + x / 2] = ave;
            newValues[offset + x / 2 + width / 2] = diff;
        }
        //Little bit of closure magic here
        int isDone = Interlocked.Decrement(ref
remainingRowsPlusOne);
        if (isDone == 0)
        {
           // Debug.WriteLine("Final calc was in loop.");
            done.Set();
        }
```


```
    };

    for (int y = 0; y < height; y++)
    {
        int yOffset = y * width;
        Interlocked.Increment(ref remainingRowsPlusOne);

ThreadPool.QueueUserWorkItem(horizontalTransform,yOffset);
    }
    int isDoneAlternative = Interlocked.Decrement(ref
remainingRowsPlusOne);
    if (isDoneAlternative == 0)
    {
        //Debug.WriteLine("Final calc occurred after loop.")
        done.Set();
    }
    done.WaitOne();
}
```

Listing 3.

```
class Program
{
    static void Main(string[] args)
    {
        int[] nums = new int[100];
        for (int i = 0; i < 100; i++)
        {
            nums[i] = i;
        }
        Random r = new Random();
        WaitCallback func = delegate(object memberOfEnumerable)
        {
            int i = (int)memberOfEnumerable;
            Thread.Sleep(r.Next(1000));
            Console.WriteLine(i);
        };

        ParallelApply(nums, func);
        Console.ReadKey();
    }

    static void ParallelApply(IEnumerable enumerable,
WaitCallback function)
    {
        using (ManualResetEvent done = new
ManualResetEvent(false))
```

```
        {
            int doneRefCount = 1;

            WaitCallback wrappedBlock = delegate(object state)
            {
                function.Invoke(state);
                int isDone = Interlocked.Decrement(ref
doneRefCount);
                if (isDone == 0)
                {
                    done.Set();
                }
            };

            WaitCallback callback = new
WaitCallback(wrappedBlock);
            foreach (object o in enumerable)
            {
                Interlocked.Increment(ref doneRefCount);
                ThreadPool .QueueUserWorkItem(callback, o);
            }
            int isDoneLate = Interlocked.Decrement(ref
doneRefCount);
            if (isDoneLate == 0)
            {
                done.Set();
            }
            done.WaitOne();
        }
    }
}
```

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Taking Advantage of Concurrent Programming for Windows, Part 3

Learn how to prevent "premature optimization" with data caches.

Larry O'Brien

Performance is a major concern of every programmer. In fact, while it's true that professionals should concentrate on delivering user value as productively as possible, sometimes performance is the most important feature. Unfortunately, it's not uncommon for an application that runs "fast enough" in development and testing to exhibit unacceptable performance failures under load.

So it's a good thing that the industry has evolved so that "premature optimization" isn't nearly as common a problem as it used to be. But it's a bad thing that a gap has opened up between already-knowledgeable, performance-oriented programmers, who tend to stay pretty low "in the weeds" when discussing techniques, and the vast majority of programmers, who invariably have an interest in utilizing their hardware to its full capacity, but haven't necessarily mastered the techniques that maximize performance.

Ask a computer scientist what leads to bad performance and he or she will likely tell you "algorithm choice." Much academic work focuses on creating algorithms whose runtime efficiency expands at a slower order than the expansion in overall complexity. That means, for instance, creating an algorithm whose worst-case behavior expands at the rate of n^2 or n^3 as the number of available possibilities increase at the rate of $n!$ or n^{n-1}. Such complexity analysis is generally expressed using the "big-oh" notation. O (controlling order), making an algorithm whose runtime performance expands at the rate of n^{2-n} would have $O(n^2)$, since the influence of the *"-n"* portion becomes relatively small as *n* becomes large.

That said, in the real world, complexity analysis usually comes in the form of occasional "sanity checks" that indicate you're avoiding exponential runtime expansion. The use of standard data structures such as those used in the .NET **Collections** namespaces generally leads to reasonable, if not always optimal, "big-oh" library performance. It's more common for most .NET programmers that algorithmic disasters are locality disasters. You don't need to know details about Load/Store units, cache "snooping," and the rest, in order to understand how locality affects most programs: accessing data in a register or on-chip cache requires just a handful of cycles, while accessing data in system RAM is an order-of-magnitude slower, and accessing data on a hard-drive is more than a million times more expensive than an on-chip operation. Don't even think about the cost of a network message!

Level 1 data caches on the Opteron and Athlon processor families are 64K in size, organized in 64-byte "lines." To the extent that you can perform multiple calculations on data stored in a cache line, you're probably very close to top performance. To the extent that your loop requires access to main memory or, heaven forbid, the hard drive (and bear in mind that most of us use disk-based virtual memory), you're not necessarily failing in your job, but you are taking a performance hit.

The issue of locality goes to the very active debate these days about dynamic languages. Object-oriented purists feel that "everything is an object" is a very important aspect of languages like Smalltalk and Ruby. On the other hand, .NET has "value types" and "reference types" that have different semantics; when you pass a value into a function and change it, if it's a value type like an integer the original instance doesn't change, whereas if it's a reference type the original does.

Java has similar behavior with its "intrinsic" types. If "everything is an object," then values have to move in and out of main memory when they're manipulated and their lifetimes have to be monitored and handled by the garbage collector. All of these machinations have a definite impact on performance.

In Part 1 and Part 2 of this series, I used a signal-processing technique called the Haar wavelet to demonstrate multithreading. Using C#'s **float** value type, applying the technique to a 16 megapixel panorama took around 6 seconds. To demonstrate the difference that objects can have, I wrapped my float values in a **class**: the same wavelet took two minutes to complete. [please explain in a bit more detail why the class implementation is so slow… specifically what extra operations are being performed under the hood?] By changing the **class** declaration to a **struct** (and thus making the class a value type), performance went back up to a respectable 9 seconds. What was the difference? The overhead of allocating, tracking, and collecting so many small objects. As is the case with many loops that perform calculation, the life-cycle of the native type or **struct** is vastly simpler than that of a reference-type object. Since there's no possibility (within the "safe" managed context) that a native type or a **struct** is being referred to by another object, the generated code can simply reuse the same memory (in the form of local variables or the heap) as the loop executes. The same algorithm, written in Ruby, has somewhat better performance than the **class**-using C# version, but runs out of memory handling the 16 megapixel file. This is not to say that objects are bad or even that Ruby is incapable of handling large amounts of data, but only that memory round-trips have performance consequences.

Even when looping on an array of value types, locality and the cache can lead to significant differences in performance. The Haar wavelet works on pairs of adjacent pixels or stated more accurately, on floating point values corresponding to the pixel's grayscale value. The wavelet has to be applied to both the horizontal and vertical axes. As Listing 1 shows, the code is identical but for the offsets of the second value.

The cardinal rule of optimization is that your efforts must be led by measurement, not guesswork. While the profiler that Microsoft provides as part of Visual Studio 2005 for Developers is good, to get the best sense of behavior you want to use a low-level profiler such as AMD's CodeAnalyst, which is available as a free download. CodeAnalyst provides a wealth of data from the on-chip performance counters, which are much more accurate than measurements taken with instrumented code. Figure 1 shows the results of running my wavelet image processor in CodeAnalyst. Naturally, the large majority of the running time is spent within the transform delegates (whose names are a little obfuscated due to the implementation of anonymous delegates), but notice that the vertical transform takes 48 percent of the time and the horizontal version takes only 26 percent. The difference is explained by the columns showing CPU Events 0x40 and 0x41. These numbers correspond to "data cache access" and "data cache miss" and, as you can see, the vertical transform has 10 times the number of cache misses as the horizontal transform. What causes this 84% slowdown? One colleague, looking at the result, guessed that it had do with cache line density: the ratio of needed data to not immediately necessary data brought into a single cache line. This is a common issue with object-oriented data structures: you just need a single integer or floating point value from each object in a collection, but the whole object (with all its other references and values and inherent overhead) is brought in as well.

In this case, all of the floating point values were stored in a single-dimensional array, so the cache density shouldn't differ between the two algorithms. Instead, the problem is simply size: in the horizontal transform, one single-precision floating point value is highly likely to be immediately available while the adjacent "vertical" value is a row's length of values away. In order to generate

calculation times of several seconds, my test data uses very large (16 megapixel) panoramic images, making it far more likely for there to be significant churn in the vertical transform.

"Solving" this characteristic would require restructuring the data, which would probably introduce more overheard than would be gained by the improved locality. On the other hand, with concrete measurements in hand, one can think about increasing the complexity of the algorithm in order to gain a performance boost; for instance, it's conceivable that by doing all of the horizontal transforms at once, and then reordering the data, and then doing all of the vertical transforms, and then reestablishing the original data order… it's a long-shot, but it might result in an overall speedup. From a practical standpoint, the crucial points about a performance-oriented .NET hotspot are:

A. Avoid unnecessary memory allocations
B. Favor value-types over objects.
C. Use threads to distribute the processing across cores and processors
D. Seek cache locality

A sequence of code that follows these rules is more likely to be efficient. Nothing, though, can overcome algorithmic inefficiency and "Big O" vulnerabilities. You have to rely on your between-the-ears optimizer to watch for those.

Larry O'Brien is a recognized expert on .NET, and is a frequent writer and speaker on software development.

Listing 1.

```
WaitCallback horizontalTransform = delegate(object state)
{
   int offset = (int)state;
   //Note x is being stepped 2 at a time
   for (int x = 0; x < width; x += 2)
   {
      float v0 = original[offset + x];
      float v1 = original[offset + x + 1];
      float ave = (v0 + v1) / 2;
      float diff = v0 - ave;
      newValues[offset + x / 2] = ave;
      newValues[offset + x / 2 + width / 2] = diff;
   }
   int isDone = Interlocked.Decrement(ref remainingRowsPlusOne);
   if (isDone == 0)
   {
      done.Set();
   }
};
…
WaitCallback verticalTransform = delegate(object state)
{
   int y = (int)state;
```

```
    int yOffset = y * width;
    int nextYOffset = yOffset + width;
    for (int x = 0; x < width; x++)
    {
        float v0 = original[yOffset + x];
        float v1 = original[nextYOffset + x];
        float ave = (v0 + v1) / 2;
        float diff = v0 - ave;
        int yIndexAve = y / 2 * width;
        int yIndexDiff = (height / 2) * width + yIndexAve;
        newValues[yIndexAve + x] = ave;
        newValues[yIndexDiff + x] = diff;
    }
    int isDone = Interlocked.Decrement(ref remainingColsPlusOne);
    if (isDone == 0)
    {
        done.Set();
    }
};
```

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Taking Advantage of Concurrent Programming for Windows, Part 4

Learn how concurrent programming can affect performance

Larry O'Brien

Other articles in this series:

- Taking Advantage of Concurrent Programming for Windows, Part 1: The Simplest Multi-Core Parallelism That Could Possibly Work
- Taking Advantage of Concurrent Programming for Windows, Part 2: Multi-Core Programming in .NET
- Taking Advantage of Concurrent Programming for Windows, Part 3: Locality Effects in .NET Multi-Core Programming

The cardinal rule of all performance programming, including concurrent programming, is that you must be guided by objective measures of performance. This rule derives from the recurring lesson that programmers will frequently guess wrongly about the cause and location of performance hotspots. This bitter truth affects all of us, even those of us who (ahem!) are supposed to know better. When I sketched out this series emphasizing easy scenarios for concurrent programming, I planned on writing a media-processing plugin for the final article. Media files are, after all, ubiquitous and big and, frankly, a little more eye-catching than a speeded-up piece of matrix math.

To make a long story short, after downloading SDKs for several audio players, video editors, and frameservers, and writing plugins for all of them, I was unable to get a convincing "win" on my dual-processor system. Details naturally varied, but the task was made harder in all of them by the callback model of plug-in processing: a relatively small chunk of data is passed to the plug-in module. In several video editors and frameservers, the main application was itself multithreaded and thereby already exploited multi-core/multiprocessor systems. In audio processing, the issues were two-fold: decoding an audio file and serving it to plug-ins is hardly a stress to a dual-Opteron system and the chunks of data passed in to the callback plug-in are so small (often 1K or 2K bytes) that the overhead of distributing the calculation erased the advantage of concurrent processing. Of course, if your algorithm is complex enough, you might see a benefit with even these small data chunks.

So, the good news is that if you want to write a video or audio plug-in that takes advantage of multiple cores or processors, you're good to go. The bad news is that you'll have to come up with a plug-in that actually does something new, not just take advantage of multiple cores!

Instead of using my word budget discussing experiments with marginal results, I thought I'd try to address some concurrent programming topics that, while not warranting a complete article in and of themselves, might be of interest. Because the topics are wide-ranging, fashioning transition paragraphs between the topics might cause more confusion than clarification, so instead the topics are separated simply by a horizontal rule.

As has been discussed throughout this series, all mainstream object-oriented languages use a shared-memory model for storing instance data. However, in the .NET CLR, there are two routes that can be taken to attain thread-local storage, that is, instance data that is specific to a particular thread. The easiest and most performant route is to mark a static field with the **[ThreadStatic]** attribute as shown in Listing 1. (Note: if you apply the **ThreadStaticAttribute** to a field that's not **static**, you will not get any kind of compile error or warning; it will just fail silently.)

Listing 1

```
using System;
using System.Threading;

class TSDemo
{
    private static int counter;

    [ThreadStatic]
    private static int threadLocal;

    static void Main(string[] args)
    {
        Thread[] t = new Thread[2];
        for (int i = 0; i < 2; i++)
        {
            t[i] = new Thread(new ThreadStart(ThreadRun));
            t[i].Start();
        }
        t[0].Join();
        t[1].Join();
    }

    static void ThreadRun()
    {
        //N.B.: Thread unsafe(see article)!
        threadLocal = ++counter;
        //Pause first thread, to allow second thread to do initial
output
        if (threadLocal == 1)
        {
            Thread.Sleep(1000);
        }
```

```
        Console.WriteLine("Shared memory value (counter) is {0}, thread
local memory
(threadLocal) is {1}", counter, threadLocal);
    }
}
```

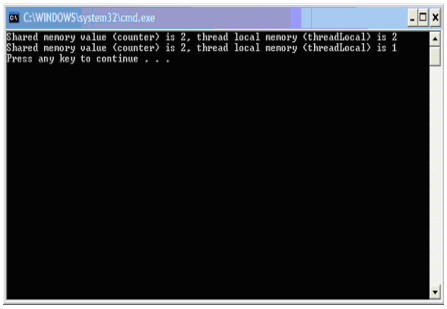

Figure 1.

As you can see, the class **TSDemo** has two **static** fields, **counter** and **threadLocal**, the latter of which has been marked with the **ThreadStaticAttribute**. The first line of **ThreadRun()** assigns to both of these fields (and introduces a race condition, which we'll discuss in a moment). If the **counter** field is valued at 1 (which it will be after the first thread executes the assignment line), that thread is put to sleep for a second. Finally, the value of both fields are output. As shown in Figure 1, the **threadLocal** field, although declared **static**, will maintain a value to the specific thread in which it is assigned.

The code in Listing 1 is not guaranteed to work; if the second thread happens to interrupt anywhere between the initial reading of the **counter** field and the check of that field to trigger the **Sleep()** (see the IL dump in Listing 2), the results could differ from what we want.

Listing 2

```
  IL_0003:  ldsfld     int32 TSDemo::counter
  IL_0008:  ldc.i4.1
  IL_0009:  add
  IL_000a:  dup
  IL_000b:  stsfld     int32 TSDemo::counter
  IL_0010:  stsfld     int32 TSDemo::threadLocal
  IL_0015:  ldsfld     int32 TSDemo::counter
  IL_001a:  ldc.i4.1
  IL_001b:  ceq
```

In order to make this code threadsafe, we need to make sure that the value of **counter** remains steady between its assignment and the check. In C#, the easiest way to do that is to use the **lock** statement, which requires a reference-type value to check. In this case,

we'll lock the Array of Thread objects (Listing 3). (*Sigh—So much for my hope to write the series without once getting into the messiness of locking*.)

Listing 3

```
using System;
using System.Threading;

class TSDemo
{
    private static Thread[] t;
    private static int counter;

    [ThreadStatic]
    private static int threadLocal;

    static void Main(string[] args)
    {
       t = new Thread[2];
        for (int i = 0; i < 2; i++)
        {
            t[i] = new Thread(new ThreadStart(ThreadRun));
            t[i].Start();
        }
        t[0].Join();
        t[1].Join();
    }

    static void ThreadRun()
    {
        lock (t)
        {
            threadLocal = ++counter;
        }
        //Pause first thread, to allow second thread to do initial
output
        if (threadLocal == 1)
        {
            Thread.Sleep(1000);
        }
        Console.WriteLine("Shared memory value (counter) is {0}, thread
local
memory (threadLocal) is {1}", counter, threadLocal);
    }
}
```

The **ThreadStaticAttribute** is actually built on top of the thread-local storage capabilities of Win32, which are additionally exposed within the CLR as methods of the **Thread** class: **AllocateDataSlot()**, **AllocateNamedDataSlot()**, **GetNamedDataSlot()**, and **FreeNamedDataSlot()**. The latter 3 functions require, as you might guess, explicit

control of their lifecycle, while **AllocateDataSlot()** will free the **LocalDataStoreSlot** as appropriate.

The **LocalDataStoreSlot** capabilities are both (slightly) more complex and less scalable than the attribute-based approach (the getter/setters for the dataslot approach take a global lock in version 2.0 of the .NET Framework). Nonetheless, for completeness, Listing 4 rewrites the previous example to do so.

This series has concentrated on the mainstream CLR languages of C++ and C#. Visual Basic .NET uses the same threading model and pretty much everything I've said in this series would apply to VB.NET (pretty much for Java, as well). Recently, though, dissatisfaction has grown with the shared-memory threading model of mainstream object-oriented languages.

The association of state and behavior is fundamental to object-orientation. That is, most objects contain both methods and fields. In the shared-memory model, all threads have access to all fields of a given instance and maintaining the integrity of that access is a responsibility shared by every thread. If a single thread shirks its responsibilities, all bets are off. Further, when such corruption occurs, it is often (maybe even generally) the case that the problem manifests itself "far away" (on a different thread, in a different program location) from the defective code. Also, such corruption is often very intermittent and dependent upon hardware and resources.

Please keep in mind that serial multitasking of a single core/processor has let a lot of unsafe concurrent code run successfully, but that era is over.

Further, the mainstream threading model has a basic conflict with the promise of **virtual** calls: you can never safely make a virtual call or invoke an anonymous delegate (closure) within a critical section. Listing 5 and Listing 6 demonstrate the problem (if you don't see a problem, increase the amount of time the **TestingClass** constructor calls **Thread.Sleep()**—sometimes it takes a while for the deadlock to manifest and, on uniprocessors, you may actually see the type of long-running, coincidentally interleaving behavior that can deceive you into thinking you have no problem). In both programs, the problem is that any kind of indirect call while holding a lock allows for the possibility that the eventually-called code will start up a new thread and do something that creates a deadlock (the problem also exists if you're using function pointers in C, but indirection is so fundamental to object-orientation and framework creation that the issue is more acute).

This is a bit like the old joke that goes:

The patient says, "Doctor, it hurts every time I move my shoulder like this."

The doctor replies, "So, don't move your shoulder like that."

If the problem is mutable state shared between threads, then the obvious solution is "don't share mutable state between threads." One way to accomplish this is to use a functional language, where once a variable has been assigned a value, the value cannot be changed. Program state is "carried" in function parameters, recursion, and return values; conceptually, these map into the things that are carried on the stack in mainstream languages (although in practice the burden on the stack can be reduced using various compiler techniques, such as call-by-need evaluation). Even if you don't embrace pure functional languages, "functional style" programming, where you minimize fields in your objects and pass state via function parameters, is both easy and facilitates unit testing.

In the .NET realm, F#, which is a derivative of OCaml, is the obvious choice for exploring functional programming techniques. Unfortunately, as of October 2006, the F# compiler does not generate multithreaded code and concurrent coding relies on the **Thread** class of the Base Class Library.

Erlang is often lauded as a language that combines a small conceptual overhead with tremendous facilities for concurrence. Haskell, on the other hand, is quite popular in academia, but can take a while to learn. Scheme and LISP are of course famously flexible (although not always implemented in a way that uses their dexterity to exploit multiple cores).

The challenges from defects that arise from the common threading model are reminiscent of the challenges of defects associated with manual memory management. Just as Java and the CLR proved that automated memory management could be a very practical if not a complete solution to the problem of resource management, perhaps some fairly simple-to-use approaches will lead to similar advances in concurrent programming. The Concur project from Microsoft's Herb Sutter proposes some very straightforward abstractions: **active** objects and methods that, conceptually, execute within their own thread. These are higher-level abstractions than are provided in OpenMP and the rumor is that within Microsoft they are being studied for inclusion in other CLR-based languages.

Note: Disappointingly, the au courant scripting languages Python and Ruby implement language-level threading within their own interpreter loop. In both cases, this seems to be an implementation detail and not a restriction within the language specification. Python implementations running on VMs—at least, Jython and IronPython—use the underlying VM threading capabilities and should be able run faster on multi-core/multiprocessor machines.

The aim of this series was to show that while concurrent programming introduces a slew of new concerns that can create problems, it is not always as forbidding a territory as it's sometimes presented. Our situation as developers is that the single-core era is over, and the future of performance-oriented software development will require exploiting first, multiple-core, and then later, many cohabiting cores and processors. In C++, C, and Fortran, the OpenMP libraries are an excellent tool for evolving code. In the CLR world, the encapsulation of Win32 threading in the objects of the **System.Threading** namespace, while arguably not ideal, is certainly vastly easier than managing the Win32 calls yourself. And profilers such as AMD's CodeAnalyst are absolutely essential if you are to achieve excellent results in performance-oriented programming.

If you measure, re-factor, and keep your state to yourself, you'll be in a good position to exploit as many cores and processors as you can fit in your system.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Linux on Multi-Core: Why WAS CE Should Get Your Attention

Learn about the Websphere Open Source Appliance with Opteron inside.

by Allan McNaughton

In a world where developers are constantly pressed to deliver more functionality, in a shorter timeframe, and at less cost than before, the combination of open source software and multi-core processors is proving to be invaluable. One can simply not afford to ignore the substantial price/performance advantage offered by these technologies.

The equation is compelling: multi-core = fast and open source = free. Although the math seems obvious, selecting the right hardware and software platform is easier said than done. A misstep in the selection process could cost you time, money, and possibly your reputation. But take heart, help is on the way as industry stalwarts IBM, Novell, and AMD have teamed up to deliver a turnkey solution to multi-core open source development.

The Open Source Appliance has arrived. Just turn it on and you'll soon be developing high performance enterprise Java applications using a pre-integrated stack of IBM WebSphere Application Server Community Edition (WAS CE), SUSE Linux Enterprise Server, and an IBM LS20 blade server powered by AMD Opteron processors.

This particular stack is described in detail in this Webcast.

Join the Community

The verdict is in. Developers love the flexibility of open source software. What they don't love is all the tinkering required to get everything working together. The Open Source Appliance solves this problem. Its pre-integrated installation of WAS CE lets you focus on developing software, not infrastructure configuration.

The WAS CE is a lightweight J2EE application server built on top of Apache Geronimo version 1.0 (the open source J2EE application server project), with key features added by IBM, and available with IBM support. Apache Geronimo brings together leading technologies (Apache Tomcat, ActiveMQ, Tranql, OpenEJB, and so on) from the broader open source community to implement the complete J2EE stack.

Key features of WAS CE include:

- J2EE 1.4 certification
- Support for multi-core processors and 32/64-bit applications
- IBM WebSphere product look-and-feel
- Easy-to-use, full featured administrative console; Support for Web tier clustering and load balancing; Hot deployment and remote deployment are supported via the admin console and integration with Eclipse
- Bundled production-quality IBM Cloudscape v10.1 database; DB2, Oracle, Microsoft SQL Server, and MySQL are also supported

- Migration aids from Gluecode Standard Edition, Apache Geronimo, and Apache Tomcat to WAS CE, and from WAS CE to other WebSphere Application Server products
- InstallShield installation with small footprint download packages—WAS CE can typically be installed in less than five minutes; ISV vendors can easily embed WAS CE.

The WebSphere Application Server Community Edition is entirely free for use in any development, test, or production environment. You can reap the benefits of open source with the peace of mind that your applications are backed by IBM. IBM offers three support options—base-level support is included when you buy the bundle and there are two additional paid programs that enable you to select the support level most appropriate for your business and technical needs.

A Solid Foundation
An enterprise Java application is only as reliable as the foundation it is built upon. The robustness of the Open Source Appliance is further strengthened by SUSE Linux Enterprise Server (SLES). Backed by Novell, SLES is a secure, reliable platform for open source computing. SLES offers unmatched performance and scalability, comprehensive open source functionality, and support for 32- and 64-bit applications running on multi-core AMD Opteron processors (there are no additional licensing fees for multi-core support).

SLES is also the first enterprise platform to include a fully integrated and supported version of Xen 3.0, the emerging open source standard for virtualization services. Xen 3.0 allows consolidation of multiple workloads on a single server or allocation of a single workload across multiple servers. With Xen and SLES, server utilization levels average up to 70 percent (as compared to 20 percent without virtualization technology). Xen works in conjunction with the virtualization features of the AMD Opteron processor to ensure the highest levels of performance.

Built to Scale
The AMD Opteron-based IBM LS20 blade server is a perfect computing platform for the Open Source Appliance. The ultra-slim and powerful, blade design delivers up to 84 servers in an industry-standard rack without sacrificing server processor performance. The LS20 supports single-core AMD Opteron processors up to 2.80GHz as well as dual-core Opteron processors up to 2.40GHz. Performance of enterprise applications is further enhanced with 1MB of L2 cache per processor and an Integrated Memory Controller that allows for increased bandwidth and reduced latency between the processor and up to 16GB of high-performance DDR memory.

In addition to delivering prodigious amounts of computing horsepower, the LS20 blade server comes ready for datacenter deployments. Connectivity to the outside world is simplified with the LS20's integrated dual Gigabit Ethernet connections. This robust design supports teaming and failover for maximum redundancy. Each blade is reliably and easily connected via two redundant paths to the passive midplane and the integrated switching and power infrastructure contained in the BladeCenter chassis. The design of the BladeCenter allows for blades to be added or changed without disrupting the operation of other blades installed in the chassis.

Get Going
The AMD Opteron-based IBM LS20 blade server and the Open Source Appliance prove that getting started with multi-core open source development need not be difficult. Its pre-integrated stack of WAS CE and SUSE Enterprise Linux Server delivers exceptional performance and functionality at a price that can't be beat (starting around $6000). The Open Source Appliance is the first open source offering of its kind and allows a developer to focus his or her efforts on

creating new value-added features, not infrastructure configuration. To get in on this great deal, contact Tech Data Resellers or your AMD sales representative.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

Making Multi-Cores Count: An ISV Licensing Primer

Learn how ISV license models can support multi-core processing and virtualization.

Alexandra Weber Morales

As multi-core processors become mainstream, proving their worth in data centers and on desktops everywhere, software vendors and purchasers alike puzzle over a new problem: How to quantify a single silicon substrate that holds several general-purpose processor cores.

From the IT manager's perspective, the cost of upgrading servers shouldn't come with an unexpected software licensing penalty. Meanwhile, many ISVs want to address multi-core processors somehow, since the performance increases, while not double, do enable higher throughput with lower power consumption and heat generation. No matter which side you're on, if concurrent programming turns out to be the next wave in software development, you'll need to know two things (assuming the Zen of threading comes quickly):

1. Which licensing model makes the most sense?
2. How many processors are in this box, anyway?

Let's get to it.

Schemes and Controversies
Though it seemed avant-garde at the time, Sun Microsystems' move to sell the entire Java Enterprise System suite at $140 per employee per year, announced in 2003, turned out to be prescient and profitable, with subscriptions to the suite at more than 1 million today. The combination of subscription-based and per-named user model used in this case also is available with a subset of products for only $50. (But check the fine print here—for service providers, the entitlement document for the suite states there is a 200:1 ratio of non-employee end user to employees.)

At the other end of the public relations spectrum was Oracle's widely debated refusal to redo its pricing schemes, charging per processor core regardless of actual performance. The reasoning was simply that Oracle considered a processor core to be equivalent to a processor. This impacted any chipmaker producing multi-core chips. That unpopular stance gave out in mid-2005, with an announcement that the price for installation on a multi-core chip would be the number of cores multiplied by an Oracle-defined factor, rounded to the next whole number. The UltraSPARC T1 processor factor is 0.25, for example, while the AMD/Intel factor is 0.50, and all other current multi-core chips (such as IBM Power chips) have a factor of 0.75. Single-core chips have a factor of 1. According to a January 2006 Oracle press release, "Licensing by processor is one of only several choices Oracle provides its customers. Other options include licensing per user and per employee. Oracle also offers the option of licensing its software on a term or perpetual basis..."

In July 2006, IBM declared a similar policy for software such as DB2, WebSphere, and Rational tools based on "processor value units," to take effect in November 2006. In this model, license fees are based on the specific underlying processor and its perceived power; a dual-core IBM Power5 chip is ranked at 100 "processor value units" per core. Dual-core x86 chips from AMD and Intel get a PVU rating of 50 per core, Intel Itanium dual-core gets a PVU rating of 100 per core, and Sun T1 (up to eight cores) get a PVU rating of 30 per core. Note that IBM considers the term "core" to be synonymous with "processor." Previously, IBM had been treating dual-core x86

chips with an explicit per socket pricing model. IBM will evolve its pricing scheme over time and presumably use benchmarking tools such SPEC CPU and TPC-C to evaluate processors—but that none of this changes existing policies for IBM mainframes.

With the advent of multi-core, chipmakers such as AMD, Intel, and Sun have suggested the "per socket" approach taken by Microsoft, Sun (whose Portal Server 7 is licensed at $57,000 per CPU or per socket), BEA, VMware, and others.

Counting cores isn't the only issue, however. With virtualization, the complications will continue to proliferate.

"More than just the per-socket or dual-core issue, the bigger issue is that as computational resources move from fixed boxes to being taken on the fly, the licensing scheme of the last 15 to 20 years has to be completely rethought," says Fred Hoch, president of the Illinois Technology Association. "Say you partition a server into seven virtual machines. The technology isn't there to allow you to track that usage from a billing or licensing perspective. There's a movement afoot to address these issues, but no one has the answer."

Among the quandaries are: What's the vendor policy if consumers run software on less than the full server? Conversely, what happens when there are more virtual machines than there are processors in the server? As fluid grid computing environments that scale according to demand grow in popularity, AMD recommends ISVs allow multiple software installations on a virtualized server or computer grid. In the near future, calculations based on usage, sites or number of employees will make more sense than hardware-based licenses. In this vein, Microsoft has made available unlimited virtualization rights with Windows Server 2003 Datacenter Edition.

One More Wrinkle: Simultaneous Multi-Threading
With Simultaneous Multi-Threading (SMT), IBM Power5, and some Intel CPUs all offer the ability to run several threads simultaneously on a single core, such that instructions from more than one thread can be executed per cycle. Each core is visible as two or more "logical" processors. This gives a performance boost for multi-threaded software (not equal to an entire extra core) that varies with the vendor and type of instructions being executed. So, a four-socket system using single core Intel Xeon processors with hyperthreading technology enabled appears to have eight CPUs—it's important to realize that in that case there are only four physical processors.

Although it doesn't use the same approach as SMT, Sun's UltraSPARC T1 chip is similar in the sense that a large number of processor cores are visible on a single system. Not designed for breakneck single-threaded performance throughput, these chips are designed for heavily-threaded server processing.

The important fact to understand is that these non-traditional approaches to threaded throughput all have different performance characteristics and cannot be easily compared to previous designs. Clearly, Oracle and IBM are taking these factors into account with their performance-oriented licensing schemes.

Processor Enumeration: Choose Your API Wisely
From a licensing and cost standpoint, what's the value of a processor core? That's up to each ISV to evaluate. As said before, AMD has generically suggested a per-socket licensing scheme for

ISVs. But there's also an optimization message there as well. AMD suggests: license software by socket, and schedule threads by available cores.

In these complex times, any hardware-based licensing scheme must distinguish between a core, a processor socket and a logical processor. Here are some strategies to follow under Windows.

On newer versions of Windows (Windows 2003 Server SP1, Windows XP Professional x64, Windows Vista), the `GetLogicalProcessorInformation()` API reveals the number of logical processors, whether the processors are on the same NUMA node, and CPU cache characteristics.

On older platforms such as the 32-bit version of Windows XP or Windows Server, to detect the number and type of processors, use a combination of the `GetSystemInfo()` API and the `CPUID` instruction. `GetSystemInfo()` will return the number of processors in a data field in a `SYSTEM_INFO` structure.

The number of logical processors means varying things, depending on the system. On an AMD-based system, it represents the number of cores. On an Intel-based x86 system with Hyper-Threading Technology on (enable or disable as an option in BIOS), the value is the number of logical processors. On Intel's dual-core Pentium Extreme Edition, for example, this value is four: two logical processors per core.

To identify a CPU's brand and details on older versions of Windows, the `CPUID` instruction prevails. If you don't want to haul out your rusty (or non-existent) assembly language skills, never fear; Visual C++ has an easy-to-use `__cpuid` intrinsic for which there's a complete example on MSDN. Note that AMD and Intel completely document their implementations of the instruction and associated machine-specific features on their respective Web sites as well.

One final caveat: in a virtualized environment, the hardware could be partitioned such that one core is allocated to a virtual machine, yet CPUID may still identify the CPU as a multi-core processor. For this reason, whenever possible, software should rely chiefly on the OS-provided APIs to query the number of available processors and cores.

Capitalizing on Concurrent Programming
In the court of public opinion, enterprise vendors have faced pressure to stop charging premiums for additional cores. Arguably, the incredible loads carried by data centers and the rising energy costs of compute farms make multiple cores a necessity—and few consumers understand why that should cost extra. Many software development experts say ISVs should worry more about threading their software for greater efficiency. Considering that quad-core processors in x86 will hit the market in 2007, it's clear that multi-threading will become even more important soon.

"We see [the multi-core issue] in Fortune 500 companies—it's too new elsewhere," says ITA's Hoch. As for concurrent programming, "I think it will force an entire rewriting of the software infrastructure. It changes computing in a way that can be beneficial for companies. Who doesn't want bigger, faster, better?"

Vendor Viewpoint
These links take you straight to the major ISVs' licensing policies at press time.

- Multi-Core Processors: Impact On Oracle Processor Licensing [http://www.oracle.com/corporate/pricing/multicore_faq.pdf]
- Microsoft Multi-Core Processor Licensing [http://www.microsoft.com/licensing/highlights/multicore.mspx]
- Lotus Notes and Domino licensing FAQ [http://www-142.ibm.com/software/sw-lotus/products/product4.nsf/wdocs/notesdominolicensingfaq]
- IBM Introduces Processor Value Unit Licensing [http://www-142.ibm.com/software/sw-lotus/services/cwepassport.nsf/wdocs/pvu_licensing_for_customers]
- FAQ on VMware Dual-Core Licensing Policy [http://www.vmware.com/vmtn/dual_core_licensing_faq.html]

SIDEBAR:

The Starting Line

Are we all at the same place when we think about multi-core processors and licensing? Let's review some basic definitions.

Basic Processor Definitions
Because hardware and software vendors use a variety of terms, the word "processor" now has a fuzzy meaning. A *processor core* is the fundamental computation and execution unit of a processor. The *processor* can be thought of as the physical package that contains one or more processor cores. Closely associated with *processor* is the term *socket*, meaning the gadget on a motherboard that you plug a processor into. Some vendors, however, consider the processor as synonymous with a processor core. In such a case, there's an implicit performance assumption: for example, that a single-processor system (one socket) with a dual-core CPU is the equivalent of or better than a dual-processor system (two sockets) with single-core processors.

Basic Licensing Definitions
The following are typical licensing schemes—but these are often blended, adding to consumer confusion. Historically, the most prevalent licensing scheme in the enterprise has been the *perpetual model*, according to an October 2004 Software and Information Industry Association study sponsored by software activation vendor Macrovision. The meaning is simple: Pay once, use forever. In contrast, *subscription-based licensing* usually means an annual fee needs to be paid to keep the software working. The *seat per-machine* model assigns the license to a specific machine. The *seat per-named user* model assigns the license to a specific individual. *Concurrent user* licensing sets a limit on many users can use the software at once, often enforced with a license server. *Per-CPU* means the cost is based directly on the number of processors in a server, similar to *per core*. Finally, *usage-based licensing* is nothing new; chargebacks for CPU *time or I/O operations that have long been prevalent on IBM mainframes.*

Trends
Licensing schemes are changing rapidly. Multi-core isn't the only reason; ISVs and customers alike need to deal with other technologies such as grid computing and virtualization. One trend is a move away from seat per-machine, the leading pricing model in 2004 at 56 percent of ISVs. A year later, the leading pricing model is concurrent user at 44 percent of ISVs, according to the

follow-up SIIA pricing study published in October 2005. Meanwhile seat per-machine had dropped to 41 percent. Seat per-named user grew from 35 percent to 40 percent of ISVs in the same time frame. Also, that research found that only about one quarter of vendors tie pricing to the number of processors. Presumably, that quarter primarily comprises enterprise vendors.

Find out more about multi-threading at the **Developer Central Multi-Core Resource Center**

White Paper: Device Driver & BIOS Development for AMD Systems

Multiprocessors and Multi-Cores

A White Paper written for
Advanced Micro Devices Inc.
Camden Associates
San Bruno, Calif.
July 2006

Introduction

Device drivers and Basic Input/Output System (BIOS) code for Windows devices are as old as Windows – and each generation of hardware has presented both new opportunities and new challenges for device-driver and BIOS developers. The move from 16-bit Windows 3.x to the 32-bit Windows 95 was the first in a long line of moves -- from DOS-based Windows 98 to the Windows NT kernel was another. Along the way, device makers have accommodated new processor architectures, new device types, new buses, and new memory subsystems.

More than a decade ago, many hardware and software makers learned to create multiprocessor x86-based servers and workstations, and conquered the device-driver and BIOS challenges that multiprocessing represented. Until recently, however, developers of mobile hardware and software – whether in notebook or laptop formats, or in a handheld configuration – rarely saw multiprocessor devices.

The new issues the mobile industry faced involved maximizing performance in a battery-operated environment, handling sleep states and non-standard display and I/O subsystems, and low-voltage considerations. Thus, many device drivers were tuned to maximize reliability and performance in those single-processor mobile environments … and many haven't even been tested in a multiprocessor system, even in the manufacturer's own test lab.

BIOS developers have also learned to accommodate advances in processor technology, improving and extending the Basic Input/Output System to offer ever-increasing performance levels on multiprocessor and multi-core mobile, desktop and server systems.

Indeed, the advent of dual-core microprocessor technology has, for the first time, introduced multiprocessing into the mainstream of personal computing, spanning mobile, desktop and server systems. Today, high-performance computers of all shapes and sizes increasingly have dual-core processors and chipsets – and in rare occasions, might experience unexpected reliability issues with device drivers and BIOS code which has not been designed, developed or rigorously tested in a multiprocessor or multi-core environment.

As dual-core processors become prevalent in the mass market for mobile devices – and as processors with more than two cores begin to enter the market– such issues will become more urgent in that domain. However, this is an issue that spans the personal computing spectrum.

This White Paper will explore the specific challenges faced by architects, developers and testers working in a multiprocessor environment, with specific attention paid to device

driver and BIOS issues. Many of the points in this paper will be presented from a mobile computing perspective, as developers in this domain may have less experience working with multiprocessor systems. However, these technological issues are applicable to non-mobile systems as well.

Multiprocessor Development Challenges

For the purposes of this White Paper, any device with more than one processor core – that is, a Central Processing Unit or CPU – will be considered a multiprocessor device. This may be a device with more than one single-core processor, or a single processor that contains two or more cores within the silicon, or indeed more than one processor, each having multiple cores. While the hardware and performance characteristics will vary depending on the actual hardware used, the important issue is that the Windows operating system automatically scales to exploit all available CPUs, and will schedule software threads to run on all of them.

These threads will include parts of the BIOS, operating system kernel and other OS applications and utilities, end users applications, as well kernel-mode and PCI (Peripheral Component Interconnect) device drivers. Kernel threads will potentially be running on all processor cores, as will device drivers. Some drivers may be fully contained within a single core, and others may have different threads running on two or more cores. While developers can have some control over thread affinity – that is, determining which core(s) are used for which thread(s) and application(s) – generally, the Windows scheduler will determine where threads are assigned and run based on the system workload and its own scheduling algorithms.

The modern Windows kernel is fully preemptive. That is, any task running in the kernel can be preempted, or interrupted, any time the thread scheduler chooses, or when an event occurs that causes a higher-priority task to be executed. Of course, there are situations when the kernel should not be preempted: when it's handling interrupts, when a lock is in place, or when the kernel is executing the scheduler itself.

In almost all cases, Windows device drivers run at a high priority level, and when run on a single-processor system with only a single core, a lock set by the driver will prevent the driver from being preempted on its processor. However, on a multiprocessor system, this may not be the case, because events occurring on a different processor may preempt the device driver, interfere with reentrancy or concurrency of the driver's routines, or cause resource contention issues related to simultaneous thread execution.

Simultaneous Thread Execution

A separate challenge is that many developers may not realize that Microsoft provides two separate sets of Windows XP and Windows Server 2003 kernels. One set is written for single processor systems, the other for multiprocessor systems. In most ways, the kernels are identical, at least as far as device drivers are concerned. However, thread scheduling and execution is different between the two kernel versions. This may be particularly surprising for mobile developers, who may not have encountered the multiprocessor kernel in their target platforms until now.

On a single processor system, all code – kernel, drivers, applications – run on a single processor. On a multiprocessor system, the workload is distributed across all processors. The highest priority ready thread (that is, one which is not waiting for I/O or for release from a locked state) runs at all times, on one of the processors. Windows schedules the next-highest priority ready threads across the processors, depending on system workload. Some of those threads may also be running on the same processor that's running in the highest priority thread.

Because more than one thread is running at the same time, it is likely that more than one driver routine (at the same or different priority level) may be running at the same time on different processors. Also, it is very possible that different code threads from a single driver will be running on more than one processor simultaneously – and those threads may not be synchronized.

The difference between single-processor and multiprocessor systems becomes apparent when a device interrupts the processor. Initially, the IRQL (interrupt request level) for that processor is at PASSIVE_LEVEL, its lowest state where interrupts are handled by Windows.

The device's driver raises its IRQL to DIRQL, a hardware-controlled state with a high priority, to run the interrupt service routine. The interrupt service routine stops the device from interrupting again, queues the I/O response, lowers the processor's IRQL to DISPATCH_LEVEL, which indicates that processing is being handled by Windows, and exits. When the I/O is complete, the driver sets the IRQL back to PASSIVE_LEVEL again. There is no chance that there can be another device interrupt until the IRQL is set back to PASSIVE_LEVEL – essentially locking the system until the driver releases the device.

However, on a multiprocessor system, an IRQL-level lock is associated only with a single CPU or core. A separate thread running on a different core – which might be associated with that same device driver, or with a different driver entirely – might be running with its IRQL at PASSIVE_LEVEL while the interrupt handler on the first core has raised its own IRQL to DIRQL or DISPATCH_LEVEL. If a second interrupt comes through, on the second core, it could disrupt the I/O, deadlock resources, or overwrite memory.

The solution is to issue explicit or implicit locks that run across multiple processors – an extra step that a mobile device developer may not have even considered taking. Even if a lock was written, if the device and driver was not tested in a multiprocessor environment, the lock code may never have been tested, debugged or tuned. This could affect synchronization and run order.

Synchronizing Access, Enforcing Run Order

Whether a device driver is running on a single processor core, or on a multiprocessor or multi-core system, it is essential that only one thread at a time can access critical data, such as data that might be overwritten by the device driver or by another process. It is also important that, when multiple threads require access to that data, the threads access that data in the proper logical order. Also, a set of operations performed on data must be performed as a single unit (atomically), without the possibility that the data might be accessed or modified by another thread before that set of operations is complete.

When the device driver is running on a traditional mobile system – with its one single-core processor – it is unlikely that synchronization issues or run-order issues will appear. However, they could easily occur in multiprocessor or multi-core systems. Indeed, without synchronization, read or write operations performed by drivers running in different thread contexts could be interleaved or happen in the wrong order, or be interfered with during an atomic set of operations. This could cause a driver to use stale or out-of-date data, or to corrupt another driver's data. This might cause a driver or system crash – or create more subtle errors that go undetected.

In some cases, these problems may be introduced by the optimizing compiler, which can aggressively reorder some instructions to improve performance – or even eliminate some instructions that it believes are redundant (but which the developer placed there as a safeguard against synchronization issues). A solution is to use the volatile http://msdn2.microsoft.com/en-us/library/12a04hfd.aspx keyword in C/++, which instructs the compiler that the variable associated with that keyword might be changed by something outside the context of the current thread. The compiler then rereads that variable each time it is referenced, and writes the value of the variable to memory each time it is assigned, thereby safeguarding against most unwanted side effects.

Developers should also consider the use of the Windows InterlockedXxx and ExInterlockedXxx routines - http://www.microsoft.com/whdc/driver/kernel/locks.mspx which perform common arithmetic and logical operations atomically, using high IRQL levels to ensure that these operations may not be preempted. These routines are designed for high performance, and are recommended for use in device drivers.

There are a number of Windows options for thread locking to guard against preemption, synchronization and run-order issues; this Microsoft paper http://www.microsoft.com/whdc/driver/kernel/locks.mspx discusses many of them. Another paper, "Multiprocessor Considerations for Kernel-Mode Drivers" http://www.microsoft.com/whdc/driver/kernel/MP_issues.mspx goes into more detail; it is focused on desktop and server systems, not mobile devices, but many of the same principles apply.

PCI Bus Device Development Notes

Many devices in a mobile system are connected via a Peripheral Component Interconnect (PCI) I/O bus. In practice, in an AMD64 system, the PCI bus is connected to the high-bandwidth HyperTransport bus via a PCI-HyperTransport adapter chip. Because device driver code running on multiple processor cores can be accessing a PCI device at the same time, and in different contexts, PCI drivers should be developed and tested to run safely on a multiprocessor system. Again, this is a specific issue for mobile developers; desktop and server device drivers have long been exposed to a multiprocessor environment, but this is new territory for mobile device developers.

When developing a PCI device driver, there are several methods of accessing the PCI bus to determine which devices are currently connected. Some of these methods are legacies from older versions of Windows, such as the old Windows 4.0 code base, and others may be limited to specific Windows builds, such as 32-bit Windows only. These methods may work in some or most cases on current versions of Windows XP or Windows Server 2003 in multiprocessor environments – but also may not perform reliably under all circumstances, and may not work properly with the multiprocessor locking methods described above. They may not also work with future versions of Windows, such as Windows Vista or Windows Server "Longhorn."

One specific challenge is when developers obtain configuration and location information about the PCI bus and its attached peripherals – such as the BusNumber, DeviceNumber and FunctionNumber of a device – by scanning the bus and using the HalGetBusData and HalGetBusDataByOffset API calls, which returns PCI port information directly.

However, beginning with Windows 2000, the Hardware Abstraction Layer APIs, including those mentioned above, are depreciated, and should not be used. Instead, developers should obtain all PCI bus and device information by asking the operating system, not by interrogating the bus or device directly.

According to Microsoft, the driver gets its resources from the Plug and Play (PnP) manager in its IRP_MN_START_DEVICE request. Typically, a well-written driver should not require any of this information to function correctly. If for some reason the

driver requires this information, the code sample to follow shows how to get the resources. The driver should be part of the device's driver stack because it requires the underlying physical device objects of the device to send the PnP request.

Normally, requests for PnP I/O Request Packets are sent at the PASSIVE_LEVEL. However, Microsoft explains that to access the configuration space at the DISPATCH_LEVEL, send an IRP_MN_QUERY_INTERFACE at PASSIVE_LEVEL to get the direct-call interface structure (BUS_INTERFACE_STANDARD) from the PCI bus driver. Store this in a nonpaged pool memory (typically in DeviceExtension). Then, call the SetBusData and GetBusData to access the configuration space at DISPATCH_LEVEL. Note that the PCI bus driver takes a reference count on the interface before it returns, so the code must dereference the interface when it is no longer needed.

Note that on all systems – including servers, desktops and mobile devices – PCI bus numbers should be considered to be transient, and can change at any time. That is another reason why developers should not rely on the bus numbers, or use the bus number to access the PCI ports directly; if a number changes, this could lead to a system failure.

Microsoft has more information on this issue, including code samples, at “How to get the configuration and location information of a PCI device” http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q253232 .

BIOS Notes for Multiprocessing

BIOS designers, developers and testers, like device-driver creators, must be aware of the issues regarding multiprocessor and multi-core systems. Those issues include all of the ones listed earlier, such as for simultaneous thread execution, as well as well-known threaded development challenges such as managing deadlocks and race conditions. There are specific issues, however, that BIOS developers must address when creating new BIOS software, or adapting existing device BIOS code to work with a dual-core or multicore chip.

If the BIOS might even need to access a potentially shared resource, it is essential to ensure that only one core or processor can ever access the resource, at any one time, from start to finish, whenever possible. If it is necessary that multiple cores or processors need to access the same resource, the BIOS coder must ensure that all of the accesses are done serially, to prevent unwanted side effects. For example, if multiple threads need to access CMOS space – and there’s a possibility that those threads might run in different cores or processors – simultaneous access might let one application processor (AP) overwrite the index value that the bootstrap processor (BSP) has written earlier. Subsequent accesses by the AP or the BSP would then retrieve the wrong data.

It is also important that the local Advanced Programmable Interrupt Controller (APIC) for each core has an enabled entry in the Advanced Configuration and Power Interface (ACPI) MADT table. (The MADT, or the Multiple APIC Description Table, describes all of the resources which can initiate or respond to interrupt.) Each processor and core must be declared using the Processor statement in the MADT. Similarly, each core's APIC must have a Processor entry in the system's Multiprocessor Configuration Table. Refer to the Advanced Configuration and Power Interface Specification http://www.acpi.info/spec.htm and Intel MultiProcessor 1.4 Specification http://developer.intel.com/design/pentium/datashts/24201606.pdf for more details.

BIOS Notes for Systems Management Interrupt

Developers face specific issues when coding logic for the Systems Management Interrupt (SMI). If an SMI handler must access shared resources through an Index/Data pair, a best practice is to ensure that the value in the Index register is saved before writing the new Index value, and then is restored after use of the Index/Data register pair is complex. This will guarantee that if the operating system is interrupted by an SMI while accessing a shared resource, Windows will retrieve the correct value from the data register when the SMI operation is complete. Otherwise, the second SMI might overwrite the index, and the original SMI handler would never be aware of it.

Note that in a dual-core or multicore system, when a directed SMI occurs, only one core within that processor will enter the SMI handler. Do not attempt to assume which core has initiated the SMI. In order to prevent any resource-contention issues while the interrupt is being handled, the SMI handler should put all of that processor's cores into SMI handling mode for the duration of the interrupt handling process. All of the cores need to exit the SMI handler at the same time by synchronizing their use of the Return from Systems Manager (RSM) instruction.

On a similar note, in a dual-core or multi-core system, when the processor is operating in Systems Management Mode (SMM), which is a mode typically invoked whenever an SMI is asserted, such as when there are changes to the power-supply status. When in SMM, each core has access to its own model specific registers (MSRs), which describe the low-level hardware status and capabilities of the system. These registers, known as SMM_BASE, SMM_ADDR and SMM_MASK, must be programmed separately for each core, so that each core has its own private save spaces. Cache-specific settings in the SMM_MASK registers must be set to be identical.

For more information about systems management mode and SMIs, see the AMD BIOS and Kernel Developer's Guide for the AMD Athlon 64 and AMD Opteron Processor

http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26094.PDF.

Note also that the BIOS may crash the system if it tries to directly access some I/O resources directly through the Advanced Configuration and Power Interface system, instead of working through operating system calls. To prevent that, Microsoft has blocked access to some resources from the ACPI Machine Language (AML) interpreter. Microsoft lists those blocked system resources, and provides recommended means for accessing them through Windows XP, Windows Server 2003 and future versions of Windows, at I/O Ports Blocked from BIOS AML http://www.microsoft.com/whdc/system/pnppwr/powermgmt/BIOSAML.mspx.

BIOS Notes for Dual-Core Specific Resources

For application developers or systems administrators, there's no practical difference between a system with one dual-core processor and a system with two single-core processors. Although there can be some performance characteristics that vary between the two configurations, for the most part those are masked by the operating system, and developers can and should assume that the cores – no matter where they are located – operate entirely independently.

However, for BIOS developers, it is important to note that there are architectural differences between a single-core and dual-core processor. All cores have their own independent L1 and L2 caches, general register sets, local Advanced Programmable Interrupt Controller (APIC), microcode, model specific registers (MSRs) and Machine Check Architecture (MCA) registers, with the following exceptions f or dual-core or multicore processors, where the cores within a processor share the following resources:

- FID/VID Control (MSR C001_0041h)
- FID/VID Status (MSR C001_0042h)
- MC4_CTL (MSR 0410h)
- MC4_STATUS (MSR 0411h)
- MC4_ADDR (MSR 0412h)
- MC4_MISC (MSR 0413h)
- MC4_CTL_MASK (MSR C001_0048h)

Also note that each core within a dual-core or multicore processor share the same Northbridge. This means that all of the PCI registers (Function 0 through Function 3) are shared. This is why the two cores share the MC4 MSRs. The second core in a NUMA (non-uniform memory access) node must be launched before initiating a FID/VID change. The second core is launched by setting the CPU1En bit See section 10.5.4

BIOS-Initiated P-State Transitions in the BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors (http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/32559.pdf) for details.

Author Acknowledgements

***Peter Aitken**, Ph.D., has been writing about computers and software for more than 15 years. His work includes over 45 books and hundreds of magazine and Web articles. As a writer and developer, he has focused on Microsoft development tools and the Office productivity suite. He has also been a speaker at software development trade meetings and is a regular contributor to the InformIT Web site on Office topics.*

***Tyler Anderson** A Master of Science graduate in Computer Engineering, Tyler Anderson is a Design Engineer in a high tech firm developing advanced DSP.*

***Steve Apiki** is senior developer at Appropriate Solutions, Inc., a Peterborough, NH consulting firm that builds server-based software solutions for a wide variety of platforms using an equally wide variety of tools. Steve has been writing about software and technology for over 15 years.*

***Anderson Bailey** is a developer with a longstanding interest in the techniques for using code to exploit processor features. He can be reached at chip.coder@gmail.com.*

***Allan McNaughton**, a veteran developer and long-time writer, is the principal at Technical Insight LLC, a firm specializing in the composition of high-technology white papers.*

***Alexandra Weber Morales** is an award-winning magazine writer and the former editor in chief of Software Development, as well as a Webmaster, singer-songwriter, and recovering auto mechanic.*

***Larry O'Brien** is a recognized expert on .NET, and is a frequent writer and speaker on software development.*

***Justin Whitney** is a regular contributor to DevX.com and Jupitermedia. He currently lives in San Francisco, where he consults for leading high-tech firms and writes about emerging technologies.*

***Alan Zeichick** A former mainframe software developer and systems analyst, Alan Zeichick is principal analyst at Camden Associates, an independent technology research firm focusing on networking, storage, and software development.*